Creating a web of places and locations for mobile applications

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Creating a web of places and locations for mobile applications.

Master Thesis

Robert Weiser
<weiserr@ethz.ch>

Supervisor: Benedikt Ostermaier & Philipp Bolliger

Prof. Friedemann Mattern

Distributed Systems Group
Institute for Pervasive Computing
Department of Computer Science
ETH Zurich

September 30, 2008

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

This thesis presents a method on how to represent *location-based services* using the capabilities of the World Wide Web in a manner, that different parties can easily create, access and search for these services. It further shows how to access services associated with places in the proximity of a geospatial postion with a mobile device by utilizing a geospatial search engine capable of retrieving and ranking the services available in such a *Web of Places and Locations*. By enabling arbitrary individuals to provide *location-based services* to this *Web of Places and Locations* the presented solution allows for a vast variety of services neither tailored to be usable only in a predefined context nor restricted in terms of use.

Zusammenfassung[1]

Diese Masterarbeit präsentiert eine Methode um Dienste, welche mit geographischen Orten verknüpft sind, mit Hilfe der vom World Wide Web gebotenen Möglichkeiten so zu repräsentieren, dass verschiedene Nutzer diese Dienste einfach erstellen, verwenden und nach ihnen suchen können. Desweiteren erläutert diese Masterarbeit wie solche standortbezogene Dienste ausgehend von der Position des Anwenders unter Verwendung einer geographischen Suchmachine gefunden werden können. Diese Suchmachine ist insbesondere in der Lage Dienste, die Teil des *Web of Places and Locations* sind, nach ihrer Relevanz sortiert anzubieten. Da es jedem frei ist standortbezogene Dienste in diesem *Web of Places and Locations* bereitzustellen, sind diese Dienste weder auf einen besonderen Verwendungszweck zugeschnitten, noch in ihrer Nutzbarkeit eingeschränkt.

[1] A german summary is required by the department of computer science at ETH for a Master's Thesis written in English

Affidavit

I hereby declare that this master thesis has been written only by the undersigned and without any assistance from third parties. Furthermore, I confirm that no sources have been used in the preparation of this thesis other than those indicated in the thesis itself.

This thesis has not yet been presented to any examination authority, neither in this form nor in a modified version.

Place, Date

Signature

Acknowledgements

I hereby want to thank Prof. Friedemann Mattern, his research group and especially Benedikt Ostermaier and Philipp Bolliger for their support with my work in the Distributed Systems Group at the Department of Computer Science at the Swiss Federal Institute of Technology ETH Zurich.

Contents

Abbreviations

AJAX Asynchronous JavaScript and XML

DC Dublin Core

DCMI Dublin Core Metadata Initiative

DCTERMS DCMI Metadata Terms

EBNF Extended Backus Naur Form

JSON JavaScript Object Notation

MBR Minimum Bounding Rectangle

OR/M Object-relational Mapping

RFC Requst for Comments

RoR Ruby on Rails

W3C World Wide Web Consortium

WOPALO *Web of Places and Locations*

1 Overview

There has been a steady rise in the number of people always carrying a portable device on their person. These devices have become increasingly more functional over the last few years, particular ly with regard to embedded GPS receivers or integrated web browsers, which allow for continuous positioning and enable the user to stay constantly connected to the internet. Having this at hand, it has become feasible to create a web of places and locations and to link these places with arbitrary services, thus creating location-based services that are accessible at specific places. In the past there were several projects exploiting the idea of location-based services. The most well known are the Cooltown project [14, 7, 6], the GUIDE tourist guide and navigation system [8] and GeoNotes [17], a location-based information system.

1.1 Problem Description

The location-based services encountered nowadays are typically provided and maintained by professional service providers. The providers tend to tailor the services offered to a set of specific needs (e.g., an electronic tourist guide). Besides that, these services are usually only available to a small group of people and can only be used within a predefined context. An example would be a printer service, accessible only to people working in the same company. Relying on such central authorities to maintain information about location-based services has the drawback that it might not be feasible or even possible for individuals to distribute custom services. Even if it were possible for arbitrary users to create location-based services and have them maintained by a central authority, the question remains whether this approach is scalable assuming that thousands of services are to be managed.
Location-based services, as they are modelled in most systems (e.g., Cooltown, GUIDE), are made accessible at the representation of the individual places - the place is aware of the services it provides (e.g., by representing a place as web page containing the available services). This is restrictive in the sense that it does not allow different parties to create and add services to the place without changing the representation of the place itself. For that reason location-based services need to be modelled in a way that allows individuals to modify existing places and services without affecting each other.
Current systems are constrained in the way that location-sensitive information can be consumed by the user. Using RFID tags, for example, to identify a spatial location [4] demands that the user is located in the immediate vicinity of the tag. However, the information that is provided to the user can be very specific due to the limited range of these tags. On the other hand, there is a variety of systems, such as GeoNotes or GUIDE, which allow the user to be located near a certain spatial position to get location-sensitive information about the associated place. Unfortunately, these systems use WLAN antennas to determine the position of

the user and decide which information to present, and therefore the information returned may not be specific enough.
There is a certain demand for the ability to consume location-based services when located reasonably near to a place, without the need to be right in front of it. In addition, these places have to be represented by other means than spatial positions, which are inappropriate for places with large dimensions. Why not have services belonging, for example, to a forest that can be used to descry its current humidity or the population of deer?
In order to be able to consume such services, however, the places they belong to have to be determined first. Furthermore, if there are several places in the neighborhood providing locations-sensitive information, the one most relevant to the user should be prioritized over others when presenting the results to the user. One could take the scenario of a person in the aforementioned forest next to a tavern, providing services such as accommodation. Such a service will certainly be more valuable for a user in the immediate vicinity of the said tavern than for a user in close proximity to the forest in general, given that this area may cover several hectars. It therefore should be possible to search for location-based services in the vicinity of the user and for the search results to be returned in an appropriate way.

1.2 Motivational Example

Giuseppe, the owner of a small Italian restaurant has decided to provide location-based services for his restaurant. He has decided upon two services: one to display the menus available for the current day of the week and one for placing table reservations for the evening. Giuseppe uses this system to define a place representing his restaurant and creates the two services which should be available there.
Paul, working across the street, is interested in location-based services available in his vicinity. He uses his mobile device, which in turn asks the system for places and their associated services. The system detects a shopping mall and Giuseppe's restaurant in Paul's neighborhood. Since the shopping mall is very large and has a variety of presumably very general services related to it, the system ranks Giuseppe's place as its prime choice and returns the places found. Paul inspects the places found using his mobile device and decides to check out the services offered by the Italian restaurant since it is listed first. He selects the "daily menu" service to browse through the menus and as he is very pleased with what he sees, he decides to place a dinner reservation for the evening using the dinner reservation service available for the restaurant.
After having enjoyed a good meal, Paul returns to his flat. He decides to share his insight about the great meal he had at the Italian restaurant with other people. For that reason Paul creates a service to write comments for the dishes served and links it with the place representing Giuseppe's restaurant.
Three days later Paul inspects the location-based services available at the Italian restaurant and comes across the service he createad . He invokes it in order to read comments by other people about dishes he has not tried yet.

1.3 Contributions

This work presents a solution on how to enable arbitrary users to store and provide location-based services using the capabilities of the World Wide Web, without the need for a single central authority to maintain these services. It is shown how to model location-based services in such a way that different parties are able to manipulate them without interfering with one another. Besides that, this work presents a search engine which is capable of retrieving location-based services in the vicinity of a spatial position and ranking the results according to some metrics intuitive for users. Furthermore, a mobile phone client is provided, representing a simple interface to the search engine.

1.4 Outline

The next chapter covers related work which has been done in the field of location-sensitive information and spatial search engines and shows how spatially encoded data can be encountered in the web today. Chapter 3 covers the design decisions taken for building the system and the constituent components of the system. Chapter 4 describes the frameworks and programming languages used for actually implementing the system and details the main features of the individual components. Chapter 5 is devoted to a short evaluation of the search engine with respect to spatial queries (the search for places in the vicinity of a spatial position). Chapter 6 concludes this thesis and suggests how the system could be extended.

2 Related Work

This chapter covers the related work that has been done in the field of location-sensitive information and location-based services, the work conducted with regard to spatial search and spatial ranking, and shows how spatial information can be encountered in the World Wide Web nowadays.

2.1 Presenting Location-Sensitive Information

The following projects strive to bridge the gap between the real and the virtual world by providing location-sensitive information or services at distinct places in the real world.

HP Labs Cooltown Project

The HP Labs Cooltown Project is a project that tries to connect the virtual world with the real world and provides context-dependant services for real world entities. The entities model either people, places or things and each have a "web presence", a virtual counterpart. The web presence is modelled as a HTML page, therefore making it accessible to a vast variety of devices, as it only requires a web browser to access its information. The web representations of people, places and things each have a unique URL and provide access to certain services (e.g., the web presence of a printer may provide a service to print documents from provided URLs). Additionally, these representations can provide links to other entities, for instance, a user's web presence may list links to the web representation of printers managed by this individual person, or a room may list a collection of links to the things or people that are contained within it.
In order to be able to actually attain the virtual representations of real world things or places, these physical objects have attached identifiers which can be sensed directly or indirectly. Direct sensing is done with the help of beacons, which emit the URL of the virtual counterpart of the respective thing or place. Indirect sensing on the other hand is done by tagging the entities either with a passive RFID tag or with a barcode. The user then has to use the capabilities of the device to identify the tag and will get the URL of the web resource by asking a web resolver to resolve the tag. The resolver is queried automatically without a specific request on the part of the user with the help of a web browser plug-in.
Real world places in the Cooltown Project are identified by beacons or other tagging methods. Since the web presentation of these places provides a set of services among other things, they actually enable the user to consume *location-based services* when the user is within reach of a beacon or a tag.

GUIDE Tourist Guide

GUIDE is an implementation of an intelligent electronic tourist guide with the aim of flexibility; providing tourists with either guided tours or letting them explore the environment on their own. It provides information both tailored to personal and environmental contexts that can be viewed and interacted with by utilizing a handheld device with an interface similar to a web browser.
Environmental contexts include, for example, the time of day and the opening times of the attractions included in the tour. The personal context requires the personal data of the user, such as his or her technical background and age, to be able to provide context-relevant information for the individual user. Additionally, to provide the user with context-sensitive information and services (e.g., ticketing), the personal context also includes information about the user's current location, obtained by receiving location messages that are transmitted from strategically positioned base stations. These stations each form WaveLAN cells that provide the user with location and dynamic information. However, these cells can be relatively large (having a diameter up to 300m) and therefore result in a low resolution of positioning information.
The information provided at distinct spatial positions is modelled using location objects (e.g., the city's castle) which can store various attributes (e.g., opening times) and also contain hypertext links to related information. Yet, the information and services are solely provided by professional providers.

GeoNotes

GeoNotes are notes that can be attached to spatial positions. Compared with other annotation systems, GeoNotes does not rely on content created by professional providers. This content typically has a high risk of being too formal and will reflect information the provider believes to be important, not the user. Hence, GeoNotes can be created and governed by arbitrary users, allowing anyone to participate and will therefore reflect information valuable to the users. Created notes can be annotated by a user name but this is not a strict requirement.
When storing a GeoNote the current spatial position is also stored. Unfortunately the WLAN positioning done by GeoNotes provides rather rudimentary positioning data. This is the reason why each note requires a place label to be provided with the note which can be used to specify the meant position of the note in natural language (e.g., coffee machine). These labels can be shared by different users to attach more notes to a spatial position.
Since there can be several hundred notes available at a certain position, GeoNotes ranks the notes and truncates the results. Ranking is done by observing how many times a label has been used for a spatial position which also compells users to provide meaningful label names. Additionally, notes available at a position can be sorted according to user name or content to provide a better user experience. However, GeoNotes does not offer the possibility to search for notes contained in a certain region. Only notes in the perceptual space of user (bound by WLAN positioning) will be listed.

2.2 Spatial Search And Spatial Ranking

Spatial search relates to the retrieval of information associated with geo-spatial positions or areas. There are comparatively few applications which allow the user to spatially search for places with associated geo-spatial areas. One of them is stated below.

Alexandria Digital Library

The Alexandria Digital Library project (ADL) [12] is a project established with the aim of developing a lightweight digital library for geo-referenced data. Geo-referenced means that data items are associated with geo-spatial regions of the earth whenever possible. The geo-spatial region types are limited to boxes, polylines or simple polygons (non-intersecting and hole-free); the spatial predicates are limited to intersection, converage and containment.
The user may then search for individual data items by providing a search mask in the form of a spatial box or a polygon. Although both the search mask and the geographic data contained in the library may be encoded as polygons, the respective minimum bounding rectangles (MBRs) are used when testing for spatial relations, and the spatial predicates are restricted to intersections. The reason for using MBRs is that spatial predicates involving polygons require specialized algorithms and are typically found only within sophisticated spatial engines. Using MBRs is both simple and mathematically efficient but it performs poorly with respect to over-estimating the area and misrepresenting the shape of a place. Besides, as spatial predicates yield only boolean results (either the search mask's MBR intersects with a place's MBR or not) the results returned need to be spatially ranked. ADL proposes a ranking based on spatial similarity using Hausdorff distances (function of the regions' size, shape and location) between two compared regions. Another algorithm, as proposed in [16] computes spatial similarity using the area of overlap and the shore factor; a factor determining how much of the area in the search mask's MBR is onshore, compared with how much of the area in the region's MBR is onshore.

2.3 Geospatial Data In The Web

Geospatial information can be found more and more frequently while browsing through the internet. The most common way is in form of ICBM adresses, that are embedded within web pages. ICBM adresses consist of a latitude/longitude-pair encoded using the WGS84 world reference system and are typically used to tag pages containing information about real world places; this concept is called "Geotagging". Wikipedia for example tags web pages like the one about Zurich[1] in a way visible for both human and machine. There are other formats which support the encoding of Geotags like for example RSS, that enables Geotagging of its individual feed items.
Other than that, projects like Wikimapia[2] present a user with a map where he or she can define places in a point-and-click like fashion. These places can be inspected by other users and can have associated information that can be edited by arbitrary users. The Wikimapia project is very popular as it is an open system and currently hosts above 8.000 000 places.

[1] *http://en.wikipedia.org/wiki/Zurich* (in the upper right corner)
[2] *http://www.wikimapia.org/*

3 Design

Location-based services, as encountered in systems like GUIDE or in the Cooltown project are created, maintained and offered by professional services providers. Having a central authority in charge to decide who to offer these services to, which parts of the services to make accessible and under what circumstances, restricts the target audience of these services to a rather small one. Besides, the maintained *location-based services* will most likely reflect content a professional service provider believes to be valuable for a group of users, not allowing content contributions from ordinary users. Furthermore, by prohibiting ordinary users from creating and publishing new services, the amount of available *location-based services* is limited to the number of services provided by the central authority. This number is both small, in comparison to a system allowing arbitrary users to create custom services, and reflects only content supported by professional service providers. Having a central authority in charge renders it impossible to make *location-based services* widely available for the public, which is why a system is envisioned that does not rely on a central authority, and is both technically open and publicly open - the *Web of Places and Locations* (WOPALO).
A *technically open* system means that all the components required to build the system are inherently replaceable, thus allowing different parties to offer and use custom components when working with *location-based services* provided by the *Web of Places and Locations*. *Publicly open* on the other hand, means that arbitrary users are allowed to define custom services and can consume any of the available *location-based services*. Since such a system will potentially attract a lot of users, *location-based services* have to be modelled in such a way that different users can create services for the same location without interfering with each other. An approach such as the one taken by Cooltown, where the representation of a place directly contains all accessible services at the very place, therefore is not applicable as it is impossible to add a service without modifying the representation of the place. *Location-based services* are therefore modelled in the form of *places* and *services* and can be brought into relation with each other by using unidirectional links; a *services* may link a *place* accessible at. In this way, users are able to reference different *services* with one particular *place* at the same point in time without the need to change the *place*'s representation. The *places* are used to represent geo-spatial areas of interest, and as we want to support a vast variety of *places* they are encoded using polygons and may be located either indoors or outdoors. Using polygons over single spatial positions, like in GeoNotes for example, has the benefit that even large *places* (e.g., Switzerland) can be represented accurately. Besides, polygons allow the representation of arbitrarily shaped *places* e.g., *places* such as an "area having a lot of bees".

To further foster the utilization of such a system and make it available to a large group of users, pre-existing and widely used infrastructures and data formats are used. The platform determined for storing location-sensitive information is the World Wide Web, as it connects people around the globe, enabling them to *access* existing web resources and/or *create* new web resources. We use HTML as the representation format for *places* and *services*, as it is

readable by both humans and machines, can present text and other multimedia types and provides hyperlinks, unidirectional links to other resources in the web. As no central authority is in charge of maintaining the *places* or *services* that are part of the *Web of Places and Locations*, the user is free to store the respective HTML pages on any host in the World Wide Web. The only piece of information required to identify such a *place* or *service* is the URL of the respective HTML page. Unfortunately, there is no standard specifying how to encode *location-based services* using HTML pages, which is why an entire section (3.1) is dedicated to describing how this is achieved by the system proposed in this thesis.

Now that *places* and *services* are created and available in the form of HTML pages hosted by different hosts in the web, it should be possible to consume *location-based services* located in the vicinity of a given spatial position. For this reason a *search engine* has been designed which is capable of searching and ranking *places* in the neighborhood of a geo-spatial position. For each of the *places* found, the *search engine* can retrieve the *services* available at this *place*, thus making it actually possible to consume a *location-based service*. Yet, as the *search engine* bases its search on *places* that have to be known beforehand, the World Wide Web needs to be indexed to gather the *places* that are part of the *Web of Places and Locations*. Besides, as a user will typically want to consume *location-based services* while walking in the streets, an application for smaller, mobile devices has been designed. This application can be used to list available *places* with their *services* and allows a user to access them.

The crucial components of the system have already been outlined, namely: a *webcrawler*, a *search engine* and a *mobile phone client*. The *webcrawler* is responsible for indexing web resources representing *places* and offered *services*. The *search engine* can then use this index to allow the user to search for available *services*. The *mobile phone client* is a frontend for the *search engine* providing a comfortable way for the user to actually consume *location-based services*. In addition to these components there is a *frontend* which facilitates the definition of new *location-based services*. The components are depicted in figure 3.1 and are covered more thoroughly in the following sections.

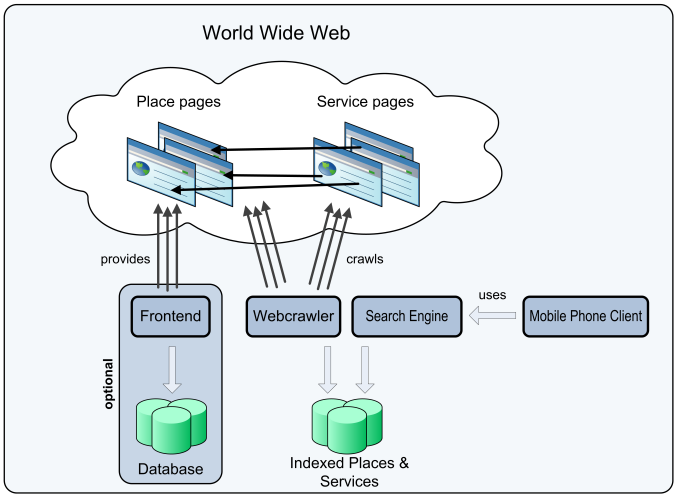


Figure 3.1: System Overview

3.1 Places and Services

As already indicated, *location-based services* as proposed by this thesis consist of *services* and *places*. A *place* is used to model a geo-spatial area while a *service* can model anything. A *service* can link to a *place*, which will make the *service* available at this very *place*. Separating *places* from *services* has the advantage that a *service* can be available at several *places* and that different *services* can link to the same *place*, thus enabling different parties to share *places*.
In order to make *location-based services* accessible over the internet, *places* and *services* have to be encoded using an adequate representation format. The chosen representation format is HTML as it has the benefit of being a W3C standard, as well as being widely used. Additionally, it is clear how to render HTML even by smaller integrated web browsers. *Places* and *services* are represented using dedicated HTML pages called *place pages* and *service pages* respectively. The URL of such a HTML page uniquely identifies a *place* or *service*. Besides embedding *place* or *service* relevant data, the HTML pages may contain any other data (e.g dynamic *service* data).
Two problems arise when using HTML pages to define *places* and *services*. The first problem relates to embedding structured data of *places* or *services* in HTML pages. A *webcrawler* should be able to extract this data without too much effort. The appearance of the web page should be influenced as little as possible and the HTML page should still be valid after embedding *place* or *service* relevant data. The other problem concerns the method of embedding geospatial data of *places* along with other *place* data into the web page.

In the following sections the *place* and *service* models are explained and afterwards it is shown how to embed structured and geospatial data into a HTML web page.

3.1.1 Place Model

A *place* can be located indoors or outdoors and may represent either a real world place such as a building or a park or it may represent an informal region of interest defined, for instance, by specific characteristics, such as: "the area were a lot of mushrooms grow". Since this poses no constraints on the form or extend of *places*, nothing can be said about their hierarchical structure, i.e., it is not always clear who should be the parent of a *place*.
Various location models [5, 1] exist for the modeling of the physical environment. Generally, these models can be categorized [10] as *symbolic location models* and *coordinate location models*. *Symbolic location models* use symbolic names for *places* and organize these *places* in a hierarchy. The location of a *place* can be determined by following the *place*'s parents since links in the hierarchy are used to model containment (a *place* is contained by its parent). *Coordinate location models* on the contrary use a global geographic coordinate system to model the geo-spatial area of a *place*. Positions in this coordinate system are usually encoded in the form of a latitude/longitude pair using the WGS84 world reference system. WGS84 defines a coordinate system for the earth and is the most widely used global positioning format, commonly used by GPS devices when returning a spatial position.
The *location model* effectively used is a *hybrid location model*. Unfortunately, existing *hybrid location models* as proposed in [2, 13] are unsuitable for this system. In the context of this work, the geo-spatial area of *places* is modelled using WGS84 as the coordinate reference system. *Places* can consist of one or more polygons encoded as a collection of WGS84

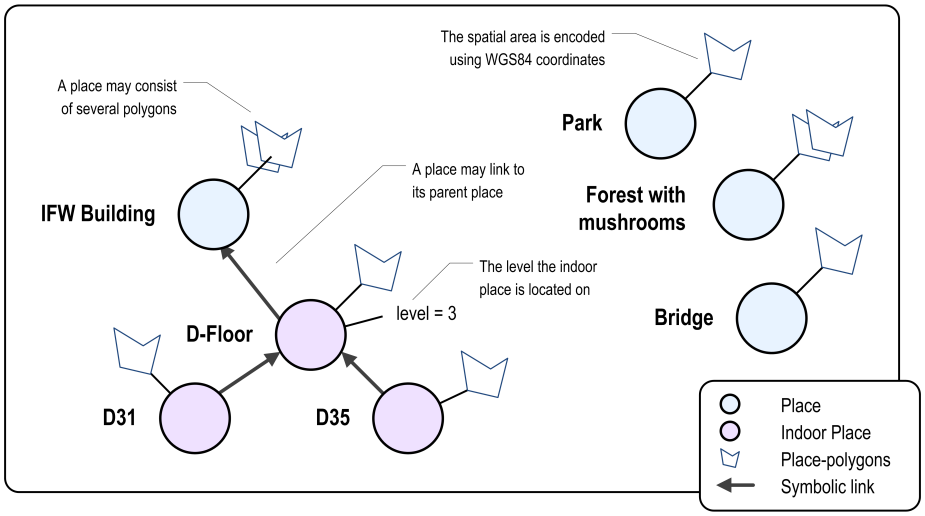


Figure 3.2: Hybrid location model used

coordinates. The benefit of using a global coordinate system is that a *search engine* can easily query for *places* in the neighborhood of a spatial position; a *spatial query*. Since *places* can be situated indoors, each *place* can symbolically reference a parent *place*. This is used to determine the floor a room is located on or the building a floor belongs to. Other than the parent reference indoor *places* always have an associated integer value specifying the level of the building they are on. Figure 3.2 depicts the hybrid location model used.

A *place* has other properties besides its geo-spatial area (e.g., a name) to make a user's interaction with it more natural. The properties are:

Name The name of the *place*

Description A short description of the *place*

Geospatial Area Represents the geo-spatial area of the *place*

Parent (Optional) Each *place* may have exactly one parent *place* assigned. When defining a *place* that is located indoors, this property has to be set to identify the floor or the building, in which this *place* is contained. The property value is the URL of the parent's *place page*.

Level (Optional) The level property is used to identify indoor *places*. It is an integer value which defines the level a *place* is located on in a building. If the level of the *place* can be determined through its parent, then the property value can be omitted.

3.1.2 Service Model

As *service pages* are ordinary HTML pages they can represent any type of information. These web pages should be adapted to be viewable with different clients (e.g., with an integrated browser of a mobile phone). The HTML page might display static or dynamic content like, for example, timetable data for a train station. One could even think of content requiring user

input like a *service* which could be used to book a table in an Italian restaurant.
As already noted, *services* reference the *places* they are accessible at. The *service* allowing a person to book a table, as mentioned above, would reference the *place* defined for the Italian restaurant.

A *service* has the following properties:

Name The name of the *service*

Description A short description of the *service*

Place References A list of *place page* URLs defining the *places* this *service* should be accessible at

3.1.3 Embedding Structured Data In HTML

Defining new *services* and *places* should be simple. As already mentioned *services* and *places* are represented using dedicated HTML pages called *service pages* and *place pages* respectively. Since there are many existing HTML pages it should be possible to turn them into *place* or *service pages* by embedding the respective metadata without to much effort.
Unfortunately it transpires that most methods for embedding structured data inside HTML pages have certain limitations, thus rendering them inappropriate for representing *places* or *services*.

The possibilities considered are stated below, together with their advantages and disadvantages.

Microformats Microformats[1] are a way to express more specialized information within the structure of a HTML or XHTML page. They piggy-back metadata onto existing HTML elements mostly using the `class`, `rel` or `href` attributes. An example of a vCard[2] [9] embedded using the hcard microformat can be seen in listing 3.1.

```
<div class="vcard">
  <span class="fn n">
    <span class="given-name">Robert</span>
    <span class="family-name">Weiser</span>
  </span>
  <a class="email" href="mailto:weiserr@ethz.ch">weiserr@ethz.ch</a>
  <div class="adr">
    <span class="locality">Zürich</span>,
    <span class="postal-code">8032</span>
  </div>
</div>
```

Listing 3.1: vCard embedded using the hcard microformat

What is immediately apparent is that the metadata defined using the microformat is visible for both human and machine. This might be acceptable for the hcard example as the data is meant to be visible to the user, but might be inappropriate if encoding the metadata of a *place* together with its spatial area (consisting of a collection of WGS84 coordinates). In any case, the hcard microformat sample shown above is quite simple but one can imagine that the use of more comprehensive microformats could lead to

[1] *http://microformats.org*
[2] A vCard is a format for representing addressbook entries.

rather complex and deeply nested HTML code. Another problem of microformats is the flat namespace which might lead to problems if embedding several microformats on a single page.

To sum up, the use of microformats, with their inability to enable the clear separation of metadata from other data and the possibility of namespace collisions disqualifies them for encoding a *place* or *service*.

XML in XHTML Another way of encoding a *place* or *service* is to represent it using XML and embedding this XML inside an XHTML web page. One restrictions of this approach is that the user has to use an XHTML page for the *place* or *service page* as it is not possible to embed arbitrary XML code inside a HTML page without invalidating it. The XHTML would have to be well-formed to ensure that a *webcrawler* visiting it could extract the valuable information without encountering problems. Moreover, it is not clear how this web page should be rendered by a client, but it is most likely that unknown XML tags would be ignored. An example of XML code embedded in an XHTML page can be seen in listing 3.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
     PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:svg="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink">
  <head>
    <title>SVG embedded inline in XHTML</title>
  </head>
  <body>
    <svg:svg version="1.1" baseProfile="full" width="300px" height="200px">
      <svg:circle cx="150px" cy="100px" r="50px" fill="#ff0000"
        stroke="#000000" stroke-width="5px"/>
    </svg:svg>
  </body>
</html>
```

Listing 3.2: SVG image embedded in XHTML

Due to the restrictions listed above this way of embedding *place* or *service* data in a web page is not used either.

eRDF and RDFa Both embeddable RDF (eRDF) and Resource Description Framework attributes (RDFa) allow existing RDF data to be embedded into a web page using ordinary HTML elements. An example can be seen in listing 3.3, which shows a HTML document with embedded FOAF metadata[3].

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
     PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns:foaf="http://xmlns.com/foaf/0.1/">
  <head>
    <title>Mark Birbeck's site</title>
    <link rel="foaf:primaryTopic foaf:maker" href="#me" />
  </head>
  <body>
```

[3]FOAF terms are used to describe friends-of-a-friend relations; see: *http://www.foaf-project.org/*

```
    <div about="#me" typeof="foaf:Person">
      <span property="foaf:name">Mark Birbeck</span>
      <a rel="foaf:weblog" href="http://internet-apps.blogspot.com/">
        XForms and Internet Applications
      </a>
      <a rel="foaf:knows" href="http://www.w3.org/People/Ivan/#me">
        Ivan Herman
      </a>
      <span rel="foaf:img">
        <img src="http://www.formsplayer.com/files/pictures/picture-11.jpg"
          alt="Picture of Mark Birbeck" />
      </span>
    </div>
  </body>
</html>
```

Listing 3.3: FOAF relations described using RDFa in HTML

As with microformats, existing HTML code is augmented with metadata, but in contrast to microformats both eRDF and RDFa have the concept of namespaces. Additionally there are several tools to help with the extraction of RDF triplets from web pages. The upside of using either eRDF or RDFa is the possibility to express arbitrary RDF data in existing HTML documents and to avoid namespace clashes. The downside is the parsing overhead required to extract the respective RDF triplets and the lack of capacity to separate the data meant for the machine from the data meant for the user. These metadata encoding standards are not used either.

META Tags META tags are a simple way of encoding metadata expressible as key/value pairs inside a HTML page. Nesting of one or more META tags is not possible, which is why the value type is always a string. An example can be seen in listing 3.4.

```
<html>
  <head>
    <title>Nutritional Allocation Increase</title>
    <meta name = "DC.Creator" content = "Simpson, Homer">
    <meta name = "DC.Title" content = "Nutritional Allocation Increase">
    <meta name = "DC.Date.Created" content = "1999-03-08">
    <meta name = "DC.Identifier" content = "http://moes.bar.com/homer.htm">
    <meta name = "DC.Format" content = "text/html; 1320 bytes">
    <meta name = "RC.MetadataAuthority" content = "Springfield Nuclear">
    <link rel = "schema.DC" href = "http://purl.org/DC/elements/1.0/">
    <link rel = "schema.RC" href = "http://nukes.org/ReactorCore/rc">
    <meta name = "DC.Type" content = "Memorandum">
  </head>
  <body>
    :
    :
  </body>
</html>
```

Listing 3.4: Web page with a memorandum created by Homer Simpson

The advantage of using META tags to encode *place* and *service* relevant data is their simplicity, readability and clean separation of the encoded metadata from the rest of the web page. META tags are not visible to the user and are easily parsable by a machine even if the HTML page is not well-formed. As can be seen in the listing above, META tags are defined in the `<head>` section of a HTML page, which is why it is sufficient for a *webcrawler* to retrieve only the head of a web page.
Additionally, since using META tags to describe metadata about a web page is no new

concept, there are various tools to generate and extract metadata.

Because of their simplicity, *place* and *service* relevant data is embedded into the respective *place* or *service page* using META tags. The symbolic links (*place*-parent relations and *service-place* relations) are modelled using META links.
Organizations such as the Dublin Core Metadata Initiative (DCMI) have put quite some effort into standardizing the meaning of a commonly used subset of metadata vocabulary (e.g name, author, description of a web resource) which can be used together with META tags [15]. Because of the clearly defined semantics and the wide acceptance of these metadata terms (DCTERMS), they are used to characterize the embedded metadata. To specify that Dublin Core metadata is used within a web page, the appropriate schemes have to be included.

Listing 3.4 and 3.3 show how *place* relevant and *service* relevant metadata is actually embedded in a *place page* or *service page* respectively. Listing 3.5 and 3.6 show a sample *place page* and a sample *service page* as they can actually be encountered in the *Web of Places and Locations*.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
      "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
  <head profile="http://dublincore.org/documents/dcq-html/">
    <title>Service</title>

    <!--tell the search engines to index the current page-->
    <meta name="robots" content="index" />

    <!--include required schemas-->
    <link rel="schema.DC" href="http://purl.org/dc/elements/1.1/" />
    <link rel="schema.DCTERMS" href="http://purl.org/dc/terms/" />

    <!--service title and description-->
    <meta name="DC.title" content="__service_name__" />
    <meta name="DC.description" content="__service_description__" />

    <!--type of page-->
    <meta name="DC.type" schema="DTERMS.DCMIType" content="Service" />

    <!--uri of a place having this service-->
    <!--might be more than one-->
    <link rel="DC.relation.references" content="__uri_of_place__" />
  </head>
  <body>
  </body>
</html>
```

Figure 3.3: Definition of a *service page*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
      "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
  <head profile="http://dublincore.org/documents/dcq-html/">
    <title>Place</title>

    <!--tell the search engines to index the current page-->
    <meta name="robots" content="index" />

    <!--include required schemas-->
    <link rel="schema.DC" href="http://purl.org/dc/elements/1.1/" />
    <link rel="schema.DCTERMS" href="http://purl.org/dc/terms/" />
    <link rel="schema.WOPALO" href="http://www.wopalo.org/wopalo/1.0/" />

    <!--place title and description-->
    <meta name="DC.title" content="__place_name__" />
    <meta name="DC.description" content="__place_description__" />

    <!--type of page-->
    <meta name="DC.type" schema="DCTERMS.DCMIType" content="Place" />

    <!--place definition consisting of one or more polygons-->
    <meta name="DC.coverage.spatial" schema="WOPALO.polygon"
     content="__encoded_polygon__" />

    <!--symbolic height used to identify indoor places-->
    <!--omitted for outdoor places-->
    <meta name="DC.coverage.spatial" schema="WOAPLO.level"
     content="__symoblic_height__" />

    <!--optional parent relation-->
    <link rel="DC.relation.isPartOf" content="__uri_of_parent__" />
  </head>
  <body>
  </body>
</html>
```

Figure 3.4: Definition of a *place page*

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
  <head>
    <title>Your next connections from Zurich, Haldenegg</title>
    <!--tell the search engines to index the current page-->
    <meta content="index" name="robots" />
    <!--include required schemas-->
    <link href="http://purl.org/dc/elements/1.1/" rel="schema.DC" />
    <link href="http://purl.org/dc/terms/" rel="schema.DCTERMS" />

    <!--service title and description-->
    <meta content="VBZ Timetableinfo" name="DC.title" />
    <meta content="Departures from Zurich, Haldenegg" name="DC.description" />
    <!--type of page-->
    <meta content="Service" name="DC.type" schema="DCTERMS.DCMIType" />
    <!--place references-->
    <link href="http://places.wopalo.org/places/22" rel="DC.relation.references" />
  </head>
  <body>
     :
  </body>
</html>
```

Figure 3.5: *Service page* example

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
  <head profile="http://dublincore.org/documents/dcq-html/">
    <title>Central</title>
    <!--tell the search engines to index the current page-->
    <meta content="index" name="robots" />
    <!--include required schemas-->
    <link href="http://purl.org/dc/elements/1.1/" rel="schema.DC" />
    <link href="http://purl.org/dc/terms/" rel="schema.DCTERMS" />
    <link href="http://www.wopalo.org/wopalo/1.0/" rel="schema.WOPALO" />

    <!--place title and description-->
    <meta content="Central" name="DC.title" />
    <meta content="Zurich Central" name="DC.description" />
    <!--type of page-->
    <meta content="Place" name="DC.type" schema="DCTERMS.DCMIType" />
    <!--place definition-->
    <meta content=
      "8.54337,47.37638 8.54393,47.3763 8.54406,47.37663 8.54441,47.37663
       8.54469,47.3768 8.54436,47.37714 8.54402,47.37712 8.54387,47.37706
       8.54362,47.37699 8.54349,47.37704 8.54320049285889,47.3770808398525
       8.54318,47.37664 8.54328,47.37663 8.54335,47.37651 8.54337,47.37638"
      name="DC.coverage.spatial" schema="WOPALO.polygon" />
  </head>
  <body>
     :
  </body>
</html>
```

Figure 3.6: *Place page* example

3.1.4 Location Encoding

It is crucial to choose an appropriate encoding for the geo-spatial area of a *place*. As already denoted in section 3.1.1 this area may consist of one or more polygons represented using WGS84 coordinates. The location encoding should therefore support the definition of polygons. Additionally, the encoding has to comply with the format chosen for embedding metadata into HTML, which essentially means that encoded polygons should be representable as simple strings.

The use of simple geo-tagging methods, which are most likely to utilize ICBM addresses[4] is therefore not sufficient as they can only represent a single spatial position. Other standards like UPU S42 or OASIS xNAL[5] are able to represent real world places but using them would also be inappropriate as no self-defined regions can be encoded.
RDFGeom provides the means to represent the geo-spatial area of a *place* using the RDF vocabulary. It is possible to define polygons which have either a 2 or 3 dimensional representation. An adequate way of embedding RDFGeom into a web page would be using eRDF or RDFa, but since META tags are used for embedding metadata the RDF triplets would have to be represented within a string. The contained triplets would still be easily parsable by a machine but this location encoding would be unsuitable as it leads to META tags with unreasonably long strings. Moreover, there has been no activity around the RDFGeom project[6] since 2004.
Other than the encoding standards mentioned above, one could use KML or GML for polygon encoding. Both languages use XML as a representation format but GML is specialized to encode any type of geographic content (e.g., bridges, roads, vehicles, etc) whereas KML supports only the more primitive geometric types such as polygons, points and boxes and focuses on the visualization of the geographic information. The number of nested XML elements is high if trying to represent geo-spatial regions using one of these encoding standards. Therefore it can be fairly tedious to write and interpret a polygon encoded using GML or KML. An example of a polygon using GML is shown in listing 3.5.

```
<gml:Polygon>
  <gml:outerBoundaryIs>
    <gml:LinearRing>
      <gml:coordinates>0,0 100,0 100,100 0,100 0,0</gml:coordinates>
    </gml:LinearRing>
  </gml:outerBoundaryIs>
</gml:Polygon>
```

Listing 3.5: Polygon encoded using GML

This is the reason why GML supports various profiles, including a profile defining a smaller and more comprehensible subset of available geometry types, as well as reducing the level of nested XML elements when defining a geometric structure. One of these profiles, namely GEORSS SIMPLE, provides the means to encode the simplest geometric structures only (e.g., closed polygons, points and boxes), however those structures can be represented as a single strings.

The format actually used is based on the GEORSS SIMPLE profile of GML since it is capable of encoding a polygon within a single string. In contrast to GEORSS SIMPLE it introduces

[4] Encoding a latitude/longitude pair using WGS84.
[5] Postal addressing standard.
[6] *http://fabl.net/vocabularies/geometry/1.1/*

a third coordinate for the height, which can be set but is not strictly required. The format can easily be processed by a *search engine* and is stripped down to contain only the required information. Several examples of the format used can be seen in the listings below.

```
"8.54220807552338,47.3663707741083 8.54200959205627,47.3668285758221
 8.54023396968842,47.3665597086304 8.54071140289307,47.3659929030338"
```

Listing 3.6: 2-dimensional polygon

```
"8.54220807552338,47.3663707741083 8.54200959205627,47.3668285758221
 8.54023396968842,47.3665597086304 8.54071140289307,47.3659929030338 400"
```

Listing 3.7: Polygon at a height of 400m over sealevel

```
"8.54220807552338,47.3663707741083,234 8.54200959205627,47.3668285758221,235.7
 8.54023396968842,47.3665597086304,231.3 8.54071140289307,47.3659929030338,233"
```

Listing 3.8: An arbitrary polygon

As can be seen above, the individual polygons are represented as a list of latitude/longitude pairs separated by a colon. The format supports an additional coordinate for the height, which can be stated at the end of the string (see listing 3.7) if all vertexes of the polygon have the same height. If the height is different for all vertexes the polygon is encoded using latitude/longitude/altitude triplets as can be seen in listing 3.8. The EBNF of the location encoding format is shown below.

```
poly       ::= poly_2d | poly_3d
poly_2d    ::= point_2d, point_2d, {point_2d}+, [number]
poly_3d    ::= point_3d, point_3d, {point_3d}+
point_2d   ::= number, ",", number
point_3d   ::= number, ",", number, ",", number
number     ::= [ "-" ] , (integer | float);
integer    ::= nonzero-digit , {digit}
float      ::= integer "." digit { digit }
digit      ::= nonzero_digit | "0"
nonzero_digit ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Listing 3.9: EBNF of the location encoding format

Dublin Core provides a term which can be used to identify a spatial region of interest, DC.coverage.spatial. As Dublin Core does not define the format to encode this region, the format described above is used and characterized by setting the schema attribute of the META tag to WOPALO.polygon. For specifying the level of a *place* the DC.coverage.spatial TERM is used too, but the schema attribute is set to WOPALO.level. To use the WOPALO schema it has to be included; see listing 3.4 for how this is actually done.

3.2 Webcrawler

One of the main focuses of the system is to allow users to define arbitrary *places* and *services* by creating the respective *place* and *service pages*. These web pages can be stored on any host in the World Wide Web and if the embedded *places* or *services* should be considered by a *search engine* when performing queries, these pages need to be indexed beforehand. There is a group of programs called spiders, bots or crawlers which wander the web successively by following the META links and HTML links of known web pages. They collect the data they are interested in concerning the visited web pages and store this data for various reasons. Since *place* and *service pages* are scattered across the internet a *webcrawler* is exactly what is required by the system to gather *place* and *service*relevant information.
Crawling the internet can be very time-consuming and will probably yield only a handful of interesting results. But as one of the assumptions is that *places* and *services* from the *Web of Places and Locations* form some kind of more tightly linked overlay, links encountered on web pages will only be followed for a certain depth if not leading to any valuable information (a *place* or *service page*).

The *webcrawler* provides two crawling modes. One usable for re-crawling already indexed *place* or *service pages* and one for quick indexing of newly added or modified *place* or *service pages*. The crawling modes are described below along with the pseudocode.

3.2.1 Regular Crawl

This is the normal crawling mode which will take all *place page* and *service page* URLs stored in the index and process them as described by the pseudocode in the listing below.

```
global urls;
add_urls(known_urls);

while count(urls) > 0:
  process_url(url);

# process a single web-page
def process_url(url):
  # crawl the page and collect interesting data if the
  # crawl depth hasn't been reached yet
  if crawl_depth_ok(url):
    page = crawl_page(url);
    links = extract_links(page);

    # check if the page is a place or service-page
    if of_interest(page):
      # store the page in the index and add all links
      store_page(page);
      add_urls(links)
    else:
      # add all contained links
      add_urls(links);
```

This crawling mode will usually require some amount of time and given the fact that most of the already indexed pages will not change too often during their lifetime, there should be some delay before re-crawling the respective web pages.

3.2.2 Single Resource Crawl

This crawling mode is designed for the scenario when a single web resource should be crawled and processed. It is obvious that the *webcrawler* should be faster when only a single resource is being crawled. However, the web resource being processed might have several links (HTML links and META links) to other *place* or *service pages*, increasing the number of pages that will be crawled. In the worst case scenario, the number of crawled resources might be the same as when doing a regular crawl. To obtain the desired behavior and avoid the re-crawling of all indexed pages, only the *place* and *service pages* which have been modified in the meantime will be processed. When encountering an unchanged *place* or *service page* the contained links will not be followed. The pseudocode is shown in the listing below.

```
global urls;
add_urls(single_url);

while count(urls) > 0:
  process_url(url);

# process a single web-page
def process_url(url):
  # crawl the page and collect interesting data if the
  # crawl depth hasn't been reached yet
  if crawl_depth_ok(url):
    page = crawl_page(url);
    links = extract_links(page);

    # check if the page is a place or service-page and if
    # it has been modified recently
    if of_interest(page) and modified(page):
      # store the page in the index and add all links
      store_page(page);
      add_urls(links)
    else:
      # add all contained links
      add_urls(links);
```

3.3 Search Engine

The *search engine* is the most essential part of the system, connecting real world places with *places* from the *Web of Places and Locations*. If provided with a geo-spatial position the *search engine* can be used to query for a list of *places* in the neighborhood; perform a *spatial query*. The *places* returned are ranked according to their relevance which is computed on the basis of several criteria discussed in sections 3.3.3 and 3.3.4.
The geo-spatial position provided defines the *point of interest* and is encoded as a latitude/longitude pair using the WGS84 reference system. If the position is indoors the latitude/longitude pair is supplemented by an integer value specifying the level of the building the floor or room is currently located on. Henceforth, spatial positions encoded as latitude/longitude pairs are referred to as *outdoor positions* and positions encoded as latitude/longitude/level triplets are referred to as *indoor positions*.
To limit the number of results returned by a *search engine*query and due to the assumption that *places* further away are less interesting than *places* in the vicinity, the size of the neighborhood is limited by a *search mask* around the *point of interest*. This *search mask* is used to filter all *places* not intersecting with it, hence these *places* will not be considered by the query. Besides that, some additional filters will be applied to *places* contained in the *search mask*, depending on the queried geo-spatial position (see section 3.3.2).
Additionally, the *search engine* can be used to query for a list of *services* at a given *place*. The *services* are sorted according to their relevance, with the criteria described in section 3.3.5.
As the *search engine* is merely used for queries it utilizes the index maintained by the *web-crawler*.

In the following section the spatial ranking assumptions are discussed. From these assumptions a set of filters and spatial ranking criteria can be derived, see section 3.3.2 and 3.3.3. Afterwards additional ranking criteria for *places* are covered (see section 3.3.4) which is followed by a section (3.3.5) discussing the used *service* ranking criteria.

3.3.1 Spatial Ranking Assumptions

To be able to appropriately rank *places*, the spatial ranking criteria need to be based on a set of assumptions that correspond well with people's subjective idea of spatial relevance.
The following assumptions are used for spatial ranking:

- *Places* which are smaller and/or nearer than others are considered more important.
  This is because smaller *places* tend to provide very specific *services* and because the user is typically interested in *location-based services* in his or her vicinity.

- If the spatial position is an *outdoor position*, outdoor *places* are of prime interest; if the position is an *indoor position*, *places* located indoor are of prime interest.
  This is based on the fact that *services* available indoors or outdoors will usually be different and that the user is usually interested in the ones available either indoors or outdoors depending on his current location.

3.3.2 Place Filters

The place filters are used to filter *places* which are not relevant in the current context and are based on the second assumption. Depending on whether the queried spatial position is located indoors or outdoors, some of the *places* contained in the respective *search mask* will not be considered in the result. The *search engine* filters outdoor *places* if the position is located indoors and filters indoor *places* if the location is outdoors.

3.3.3 Spatial Ranking Criteria

The spatial ranking criteria determine the relevance of a *place* for a given geo-spatial position and are derived from the first assumption.
There is a single ranking criterion used to rank the size of a *place* which is based on its *area*. The criteria for the proximity of a *place* to the *point of interest* are based on the *minimum object distance*, the *maximum object distance*, the position of the *centroid* and the *level* of the *place* if it is indoors. All the criteria influence the overall spatial rank of a *place* by introducing a small number of penalties each. The penalty is used to penalize *places* not conforming to the spatial ranking assumptions (e.g.: large *places* and *places* that are far away from the *point of interest*). *Places* that have smaller penalties than others will be ranked higher.
The penalty values for the mentioned criteria are computed using penalty functions; the overall spatial penalty is computed by computing a weighted sum of the individual penalty values:

$$penalty = [w_{area}, w_{min}, w_{max}, w_{centroid}, w_{level}] * [\mathbf{f}_{area}, \mathbf{f}_{min}, \mathbf{f}_{max}, \mathbf{f}_{centroid}, \mathbf{f}_{level}]^{\mathbf{T}}$$

The penalty functions are unconstrained, meaning that the resulting penalty value can be arbitrary high. This is done so that if a *place* performs poorly for some ranking criterion, it will have a poor overall spatial rank (e.g.: standing directly next to a *place* covering the area of Switzerland). If one constrained the penalty functions to return penalty values between zero and one, a *place* could be ranked better in comparison with a given other one by compensating for one high penalty value with other low ones. However, this is undesirable as a *place* should not be able to compensate for being way too large or too far away, hence the penalty functions are unconstrained. The individual penalty functions are of the following form:

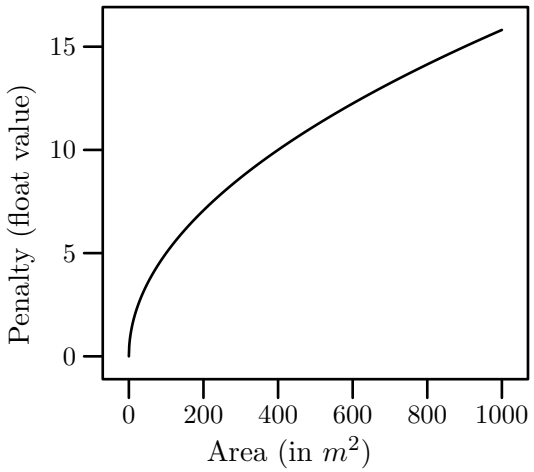
$$\mathbf{f} = f(\vec{x}, \vec{c})$$

The values computed by the penalty functions $\mathbf{f}$ depend on the queried spatial position $\vec{x}$ and a set of constants $\vec{c}$ used to tune the behavior of the penalty function and thus the resulting penalty value.
The weights w used to compute the overall spatial penalty and the constants $\vec{c}$ used in the individual penalty functions had been determined using various scenarious, see chapter 4.7.

On the following pages the spatial ranking criteria are discussed in turn. For each criterion it is explained why it was chosen and the used penalty function is depicted along with an application scenario. The curve shapes of the penality functions are based on design decisions, the constants $\vec{c}$ for the functions are merly used for tuning their behaviour. The constants for the penalty functions are set to the values that turned out best when training the individual penalty functions with the *testbed* (see chapter 4.7); the weights used are set to 1.
A preliminary evaluation of the *search engine* with respect to *spatial queries* can be seen in chapter 5.

3.3.3.1 Area

A *place* is considered to be more important the smaller it is. The reason for this is that information or services associated with a *place* tend to be more specific if it is small compared to a *place* covering a larger area. An example of this would be a tram station probably having more specific services compared than a *place* covering the region of Switzerland.



Function:

$$\mathbf{f} = \frac{\sqrt{area}}{c_1}$$

Chosen constant values:

$c_1 = 2$

Figure 3.7: Area penalty function

Penalty Function The area penalty which can be seen in figure 3.7 is invariant because it is completely independent of the proximity of the polygon to the *point of interest* or the size of the *search mask*. The penalty function grows with the square root of the area so that it will not increase too rapidly. This is done so that extending a *place* in one direction, which will increase its area by a multiple, will not influence the penalty too much.

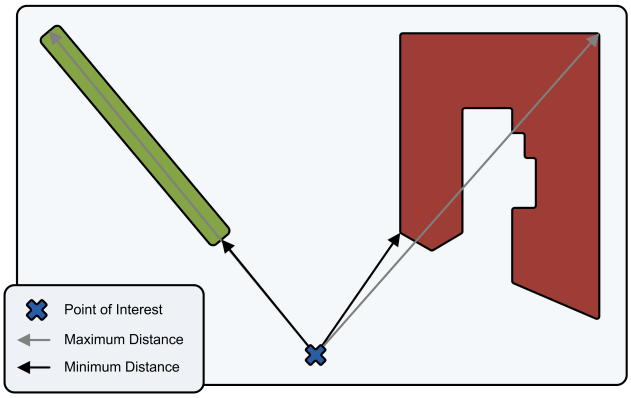
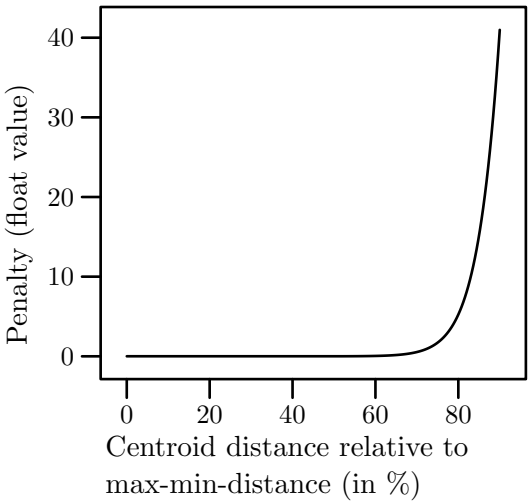


Figure 3.8: Area rank application scenario

Application Scenario Both *places* in figure 3.8 have the same minimum and maximum object distances to the *point of interest*, marked with a blue cross. The spatial ranking of the green *place* will be better as its area is smaller.

3.3.3.2 Centroid

The centroid penalty is used to penalize *places* with large perimeter and comparably small area . Usually the shape of a *place* having these properties has some sort of extension(s) (e.g., the red *place* in figure 3.10). The smaller the area of such an extension, the less important proximity to the area is.



Function:

$$\mathbf{f} = \frac{c_1}{e^{\frac{\mathbf{d}}{c_2}}}$$

$$\mathbf{d} = 100 * \frac{\overline{CB}}{\overline{AB}^{c_3}}$$

Chosen constant values:

$c_1 = 200$
$c_2 = 10$
$c_3 = 1.2$

Figure 3.9: Centroid penalty function

Penalty Function The penalty, depicted in figure 3.9, depends on the distance of the centroid from the *point of interest* compared to the minimum and maximum object distance. The nearer the centroid is to the maximum distant point the higher is the penalty.
In most cases this penalty will not influence the overall spatial penalty too much.

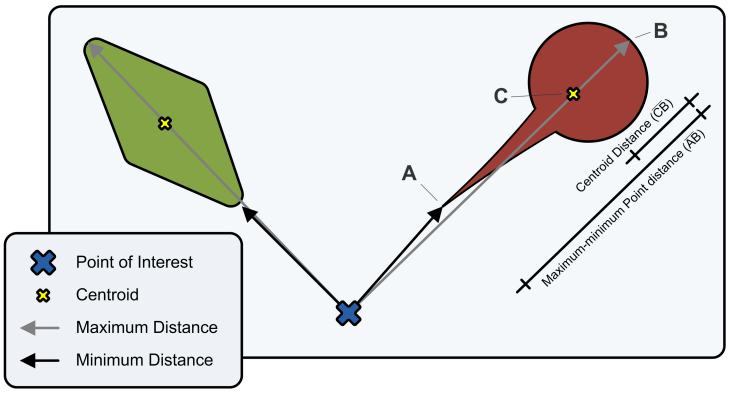
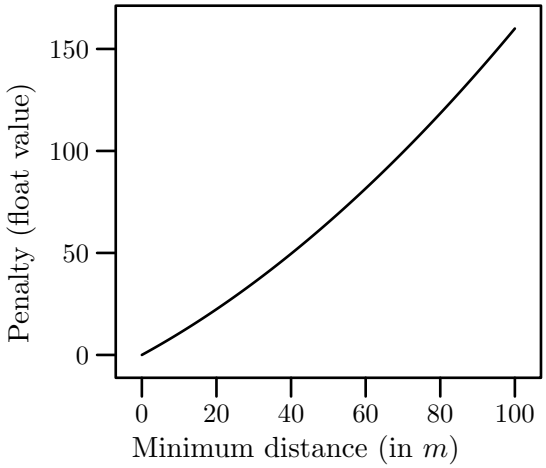


Figure 3.10: Centroid rank application scenario

Application Scenario The red *place* in figure 3.10 has a worse spatial ranking than the green one as its centroid is nearer to the maximum distant point. The area, the minimum and maximum distance of both polygons from the *point of interest* are the same.

3.3.3.3 Minimum Object Distance

This penalty depends on the distance from the *point of interest* to the nearest point of a *place*'s polygon. As expected, *places* further away are considered less important than places in the vicinity of the centroid.
If the *point of interest* lies within a polygon of a *place* the minimum distance penalty equals zero which is reasonable as one could not possible get any nearer to that *place*.



Function:

$$\mathbf{f} = min + c_1 * \frac{min^{c_2}}{mask\_radius}$$

Chosen constant values:

$c_1 = 1.2$
$c_2 = 2$

Figure 3.11: Minimum distance penalty function

Penalty Function This penalty function (figure 3.11) grows more or less linearly and additionally increases if a *place* is located nearer the *search mask* boundaries. This is done as *places* situated near the *search mask* boundaries will hardly be of interest for a reasonably large *search mask*.

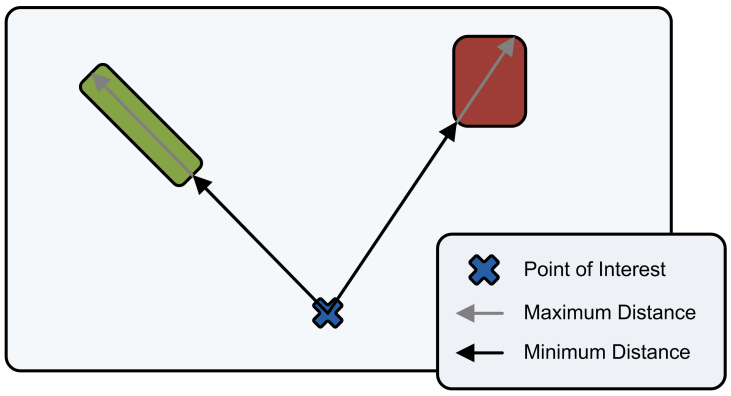
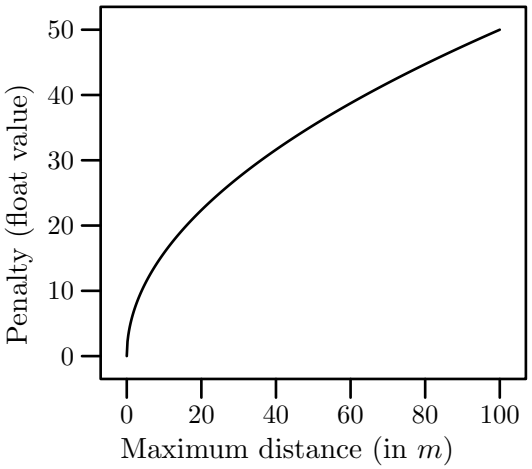


Figure 3.12: Minimum distance rank application scenario

Application Scenario The *places* in figure 3.12 cover the same area and have the same maximum distance. The green *place* will be ranked higher as it is nearer to the *point of interest*.

3.3.3.4 Maximum Object Distance

This penalty depends on the distance from the *point of interest* to the furthest point of a *place*'s polygon. *Places* which are more compact than others will get a smaller penalty. This criterion is useful to penalize *places* distributed over a lot of territory, but covering a rather small area (e.g., a long road).



Function:

$$\mathbf{f} = \frac{\sqrt{max * c_1}}{c_2}$$

Chosen constant values:

$c_1 = 100$
$c_2 = 2$

Figure 3.13: Maximum distance penalty function

Penalty Function This penalty function (see figure3.13) decreases slightly the further it is from the maximum distant point of the polygon. As a result the other spatial ranking criteria get increasingly more important, which to some extent tolerates nearby *places* having extensions in one or more directions (e.g., standing near position B in figure 3.10).

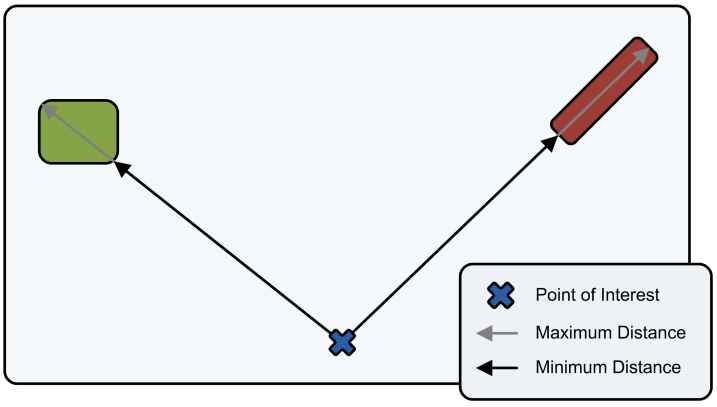
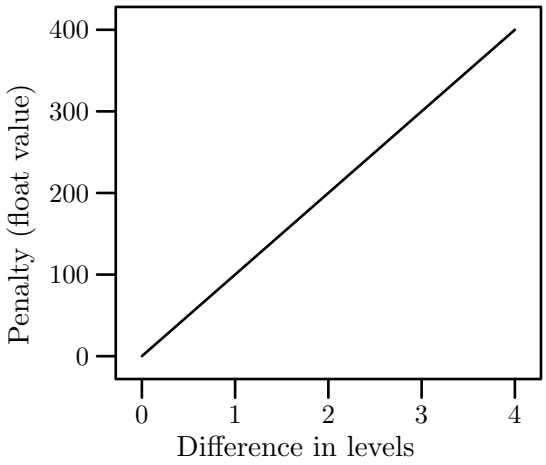


Figure 3.14: Maximum distance rank application scenario

Application Scenario Figure 3.14 shows two *places* of the same size and with same distance to the *point of interest*. The spatial rank of the green polygon is better as it more compact.

3.3.3.5 Level

This penalty applies to indoor *places* only. *Places* which are on the same floor will be preferred over *places* which are a floor higher or lower than the current one. This makes sense as *places* on the same level are nearer and therefore usually easier to reach.



Function:

$\mathbf{f} = c_1 * |level_{position} - level_{place}|$

Chosen constant values:

$c_1 = 100$

Figure 3.15: Level penalty function

Penalty Function The penalty function for the level which can be seen in figure 3.15 grows linearly as there is no reason to increase or decrease it if the level difference grows. The penalty value per level was selected in such a way that it can be compared to the penalty of a *place* located on the same floor in a certain distance away of the *point of interest*.

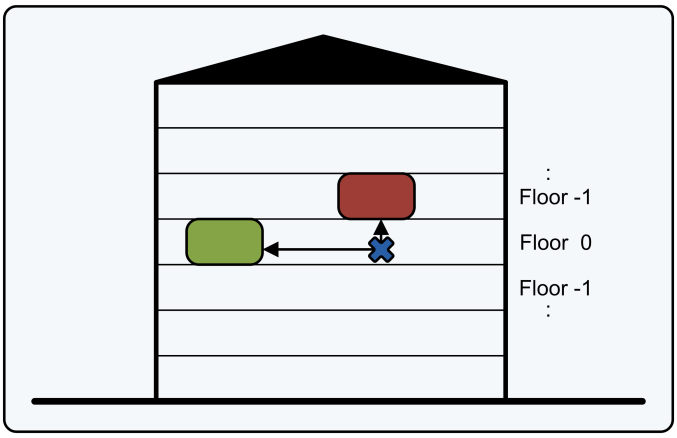


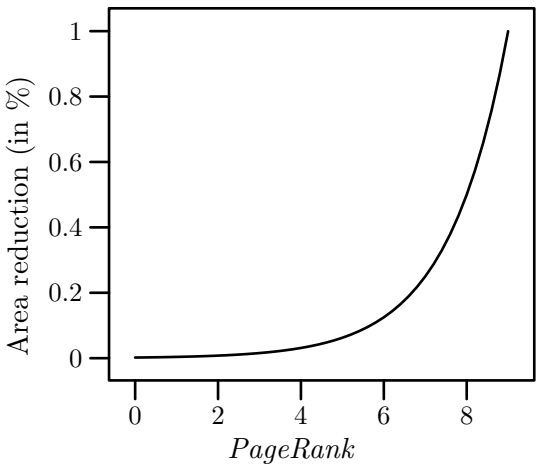
Figure 3.16: Level rank application scenario

Application Scenario The green *place* in figure 3.16 will get a better spatial rank as it is on the same floor and an acceptable distance from the *point of interest*. The size of the *places* in the figure are the same.

3.3.4 Additional Ranking Criteria

There are cases when spatial ranking alone is not enough. A good example for this would be a user located in France near the Château de Versailles, surrounded by a large number of booths selling food, souvenirs and other items of interests to the visitor. If all these real world *places* had a virtual counterpart in form of a *place page* a *search engine* query using spatial ranking would, of course, only favor all near and small *places* possibly ranking Versailles as the least important *place*.
What has not been taken into account so far is the "popularity" of a *place* which should increase its overall rank. Since it is difficult to tell which *place* is popular and which is not, the *Google PageRank* [3] of the respective *place page* is used as a metric. The *Google PageRank* is a measure determining the citation importance of a web page. If a web page contains valuable data, it will be cited (linked) by a large number of web pages spread across the internet thus having a high *PageRank*. The *PageRank* ranges from 0 to 9, with 9 denoting the highest rank. Depending on the *PageRank* a fixed percentage of the area penalty of the *place* is ignored, thus artificially shrinking the size of it. One could argue that it would be better to decrease the overall penalty of a popular *place* but this would penalize less popular *places* being in the immediate vicinity.
As already stated, this additional ranking criterion is applied after all *places* are spatially ranked and because it depends on the area penalty.

The function defining the area penalty percentage reduction to be applied is depicted in figure 3.17.



Function:

$$\mathbf{f} = area\_rank * \frac{1}{{c_1}^{9-page\_rank}}$$

Chosen constant values:

$c_1 = 2$

Figure 3.17: Advantage through high *PageRank*

A typical web page has a *PageRank* of about 5. Assuming that a typical *place page* will also have a *PageRank* about 5, the corresponding *place* should not profit too much from its rank. This is the reason for not having a linear function, but only a function rewarding *place pages* with a *PageRank* above normal.

3.3.5 Service Ranking Criteria

A *place* may be referenced by several *services* which need to be ranked as well. The ranking criteria for *services* can be mainly categorized into two classes, one based on user behavior (e.g., the number of visits to a certain *service page*) and one based on the link graph of *service pages* (e.g., *Google PageRank*). The problem with the first class of ranking criteria is that they can be easily manipulated by a user and are therefore not of any practical use.

A few criteria of both classes have been analyzed and the results are discussed below.

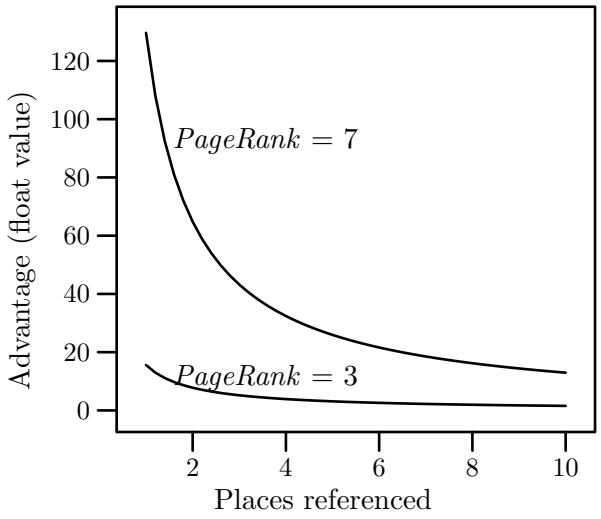
Page Visits The ranking criterion is based on the number of page visits of the *service page* since it was indexed by the *webcrawler*; a *service page* visited more often has a better rank. A disadvantage of this criterion is that newly indexed *services* have to be treated specially because of their low initial rank. One could also change the criterion to be based on the number of page visits per month or day which would be a bit better.
The main problem of this ranking criterion is that it can be manipulated easily by repeatedly visiting the same *service page* and therefore it is of limited value.

Page Follows This criterion is based on the number of page visits of a *service page* when a particular *place page* was visited beforehand. It has the same disadvantages as the criterion described above and therefore is not used either.

Page Rank This criterion is based on the *Google PageRank* of the page. The *PageRank* determines the citation importance of a web page; the higher this rank the better.

Page References This criterion is based on the number of *place pages* a *service page* references. Since most *services* are expected to be designed for exactly one *place* the rank gets worse the more *place pages* a *service page* links. This is done to avoid spamming by a single *service* that is linking a large number of *places*.

The *services* are ranked using the last two criteria as they are independent of individual user behavior. A *service* receives a certain advantage over other *services* if either having a high *PageRank* or if it references only a few *places*. As only two criteria are used, the overall rank is computed using a function combining both criteria. This function is biased in such a way that a *service page* with a higher *PageRank* will be penalized less for referencing more than one *place page* (see figure 3.18). This is based on the assumption that a page with high *PageRank* is more important and trustworthy than one having a low *PageRank* and linking a lot of *places*. Besides, as it is exponentially harder to get a good *PageRank*, *service pages* having a good *PageRank* will have an exponential advantage over *service pages* with a lower *PageRank*.



Function:

$$\mathbf{f} = \frac{page\_rank^{c_1}}{|referenced\_places|}$$

Chosen constant values:

$c_1 = 2.5$

Figure 3.18: Combined *service* penalty

3.4 Mobile Phone Client

The *mobile phone client* is a frontend for the *search engine* tailored for a specific set of mobile phones. The reason for having a frontend for the *search engine* is twofold. On one hand, a user should be able to consume *location-based services* through a minimum amount of interaction with the system. On the other hand, the geo-spatial position of the user should be determined automatically by the client.
When located in the neighborhood of a *place* a user can use the *mobile phone client* to search for it. The client can determine the spatial position by using the capabilities of a mobile phone. It uses GPS to retrieve the current position in form of a latitude/longitude pair encoded in WGS84. If GPS positioning is successful the *mobile phone client* automatically queries the *search engine* for *places* in the vicinity of the confirmed position and presents these *places* to the user. The user can then choose to view the *services* accessible at one of the returned *places*. Again, a *search engine* query is performed in the background by the *mobile phone client*. The query returns a ranked set of *services* for the selected *place* which are then presented to the user. By clicking on one of the displayed *services* the user is taken to the corresponding *service page*, thus actually consuming a *location-based service*.
It is not possible to determine the GPS position of a user at all times. The user could be located indoors and therefore beyond the reach of any satellites used by GPS. As a result, the *mobile phone client* can try to determine its position by sensing for Bluetooth devices in its vicinity. Since the *indoor positions* for some Bluetooth devices are known a priori, the *mobile phone client* can use these positions to query the *search engine* for *places* located either indoors or outdoors.

3.5 Frontend

The *frontend* is an optional component of the system and has the aim of simplifying the addition of *places* to the *Web of Places and Locations*. It displays a scrollable and zoomable map and enables the user to point-and-click to define the geo-spatial area of a *place* on the map. By clicking on the map, the vertexes of a *place*'s geographic shape are defined at the clicked geospatial positions. This way the user does not need to specify the collection of latitude/longitude pairs a *place*'s geographic shape consists of manually and can actually observe the *place* while defining it. In addition to this *place*, the *frontend* renders previously defined *places* onto the map. This is done for informational purposes and to minimize the probability of a user creating a *place* which has already been defined by another user.
The *frontend* provides the means to store the created *places*. Each of the stored *places* has its own dedicated *place page* hosted by the *frontend*, therefore making it accessible for a webcrawler. *Places* can be stored at the *frontend* as the typical user is neither able to host a *place page* nor actually wants to do so. On the other hand if the user decides to host the *place page* by him- or herself, the *frontend* can generate the required HTML code for the *place page* of the respective *place*.
The place model used (see section 3.1.1) distinguishes between indoor and outdoor *places* which can both be defined using the *frontend*. As the creation of outdoor and indoor *places* is different, they are discussed in separate sections below.

3.5.1 Defining Outdoor Places

The user is presented with a map which can be used to define the geo-spatial area of a *place*. This area may consist of one or more polygons and the user can define the vertexes of these polygons by simply clicking on the map. When the user has defined the geographic shape of the *place*, he or she has to specify its name and description. At this point the user can choose to let the *frontend* store the *place* or if the user decides to host the *place* personally, the *frontend* can output the HTML code for the *place*'s *place page*.

3.5.2 Defining Indoor Places

The *frontend* can be used to create indoor *places* that define either a room or a floor of a building. To start the definition of indoor *places* the user is required to select the building they should be contained in. The user may then add floors to the building and for each floor the user may add zero or more rooms. When defining a floor the user has to specify its name, description and most importantly its level so that it is clear that the respective *place* is located indoors. The user can then add rooms to a floor by defining their geo-spatial area on top of a map in the same manner as when defining outdoor *places* and is required to provide a name and description for the corresponding *place*.
As it is difficult to identify the vertexes of the geographic shape of a room, a user can provide a floor image which will then be rendered on top of the map. The *frontend* provides an additional view to appropriately position, scale and rotate this image. This view consists of two regions, one showing the uploaded image and the other showing a map and rendering the respective floor polygon on top of it. The regions have two draggable reference markers which can be used to position the floor image. If finished, the user can choose to save the floor image together with its position.

3.6 VBZ Service

The *VBZ service* was designed to have a useful sample *service* for the system. The idea of the *service* is to display electronic timetables for trams departing at distinct stations, i.e. trams departing from a station within the next few minutes are displayed.
To actually make this electronic timetable available at certain *places* the *VBZ service* provides several *service pages*. A *service page* renders the departures from a single tram station and references the *place page* of the corresponding tram station, therefore being available at the very *place* (e.g.: electronic timetable page for Haldenegg referencing the *place page* of Haldenegg).
Since a person consuming such a *service* will usually be located near to a tram station and carrying some sort of handheld running the *mobile phone client* (see section 3.4), the *service pages* are optimized for smaller displays.

4 Implementation

This chapter covers the implementation of the components described in chapter 3 and depicted in figure 4.1. Besides these components, this chapter also covers the implementation of the *testbed*, which was designed to train the penalty functions used by the *search engine* when doing *spatial queries*.
As several components were implemented as web applications, a separate section (section 4.1) is dedicated to the design decisions made, regarding the chosen frameworks and programming languages.

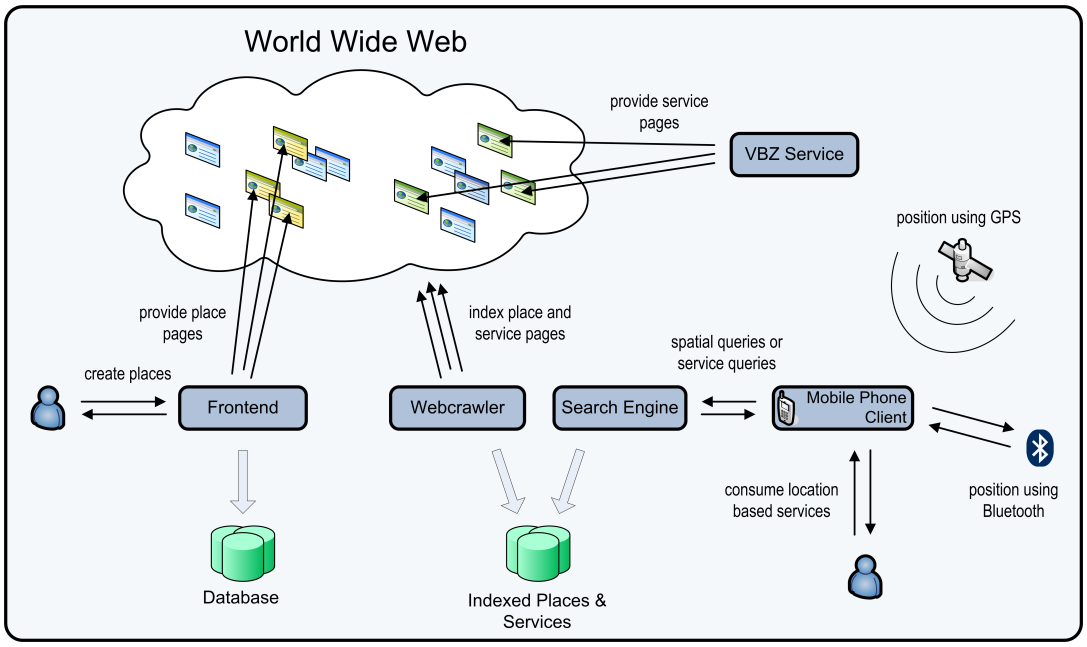


Figure 4.1: Component Overview

4.1 Used Frameworks

4.1.1 Web Applications

The *search engine*, the *frontend* and the *VBZ service* have been developed as web applications to be accessible in the World Wide Web. PHP and the Ruby on Rails (RoR) framework were the choices which have been considered for implementing the respective applications.

RoR was actually used as it provides certain advantages over PHP in terms of facilities. RoR forces a programmer to structure his program according to the MVC (model-view-controller) architecture [11]. The MVC architecture separates the application logic from the user interface and a set of model classes, resulting in an application which is easier to modify (e.g., one can modify the visual appearance of the application without affecting the application logic). RoR allows a programmer to create templates for the individual views and allows different views to share parts of these templates. Additionally, RoR automatically maps model classes to database tables (OR/M) and allows the programmer to easily store and retrieve model specific data within the application. Besides that, RoR offers the possibility to map URLs to arbitrary controllers, actions and parameters. The mapping of an URL to such a controller/action pair is called a *route* in the context of RoR. One could have used PHP together with Smarty[1] (provides view templates), Propel[2] (provides OR/M) and Apache mod_rewrite[3] (provides rewriting of URLs) to get the same behavior as when using RoR but the configuration overhead would have been much larger.
Furthermore, there exist several RoR plugins and RubyGems[4] which facilitate programming and project management when developing with RoR. The ones used for all web projects are Piston[5], which is a tool for managing rails plug-ins and the Haml plug-in[6]. The Haml plug-in is used to parse view templates written in Haml. Haml is a markup language that enables a programmer to write shorter and more comprehensible code and is used to describe the XHTML of a web document.
The web applications use PostgreSQL as their database backend. The reason for using PostgreSQL is that it can be used together with PostGIS[7], which is a spatial database extension that allows one to store geometric objects such as polygons in the database. PostgreSQL itself could have been used for storing simple geometric objects as it provides the capabilities to do so, but PostGIS has the benefit of a spatial index which can tremendously speed up certain queries. Moreover, PostGIS provides a comprehensive set of geometric methods (e.g., spatial intersection, spatial overlap) and geometric constructors.

4.1.2 Mobile Phone Client

There were two decisions to be made for the *mobile phone client*: one concerning which mobile phone to use and another concerning the programming language.
The iPhone was one of the mobile phones reviewed. The main problem of the iPhone generation available at time when the *mobile phone client* was developed, was the lack of a GPS receiver. The only way to retrieve the spatial position of a user was by means of WLAN or Cell-IDS. WLAN positioning is impossible at certain geospatial positions, whereas positioning using CELL-IDS is not very accurate. The other disadvantages of the iPhone were the small user base, actually writing applications for it, and the limited number of available libraries, being rather new at that point in time.
The other mobile phone considered and actually used was the NokiaN95 8GB. The NokiaN95

[1] *http://www.smarty.net/*
[2] *http://propel.phpdb.org/trac/*
[3] *http://httpd.apache.org/docs/2.0/mod/mod_rewrite.html*
[4] A gem is a package containing e.g., a ruby library.
[5] *http://piston.rubyforge.org/*
[6] *http://haml.hamptoncatlin.com/*
[7] *http://postgis.refractions.net/*

8GB uses the S60 Platform, which is a software platform for mobile phones that runs on top of SymbianOS. The NokiaN95 8GB has an integrated GPS receiver and can be programmed using either SymbianOS C++, J2ME or Python. Symbian OS's flavor of C++ is very specialized and has a rather high learning curve, which is also the reason why it was disregarded as a possible programming language. J2ME enables a programmer to write applications for smaller devices using a subset of the Java programming language. Due to time constraints and the need to write quite a bit of Java code even for simple programs, the *mobile phone client* was not implemented using Java. The *mobile phone client* was developed in Python as it emphasizes programming productivity. To be more specific: the *mobile phone client* was developed in Python for S60 (PyS60[8]) which is Nokia's port of the Python language for the S60 Platform. PyS60 comes with a Python interpreter that can be used to run Python scripts located on the mobile phone. PyS60 allows a programmer to use the default widgets of the mobile phone when writing applications and comes with libraries that allow one to easily perform GPS inquiries or do Bluetooth sensing.
As incremental program-and-test cycles while developing the *mobile phone client* would have necessitated copying the respective Python script to the mobile phone each time by using an USB cable, the Batoo Rapid Prototyping Environment was used. The Batoo Rapid Prototyping Environment is an application environment that can be used when writing Python scripts for the S60 platform. It allows the synchronization and execution of Python scripts located on a computer with a mobile phone. Synchronization is done by sending the code differences of the Python script to the mobile phone using Bluetooth communication. Besides that, the Batoo Rapid Prototyping Environment provides a remote console that can be used for simple debugging when remotely executing a Python script on the mobile phone. The Batoo Rapid Prototyping Environment was developed by Robert Adelman at the Distributed Systems Group of ETH Zurich.

4.1.3 Webcrawler

The *webcrawler* was developed using Java as the programming language of choice because of existing experience and because of easy integration with a database using the appropriate JDBC (Java DataBase Connectivity) driver.

[8] *http://opensource.nokia.com/projects/pythonfors60/*

4.2 Webcrawler

The *webcrawler* is responsible for indexing *place* and *service pages* available in the World Wide Web. It has been developed using Java and shares its database with the *search engine* so that the *search engine* can utilize the indexed *places* and *services* when performing queries. A schematic overview of the database layout can be seen in figure 4.3 on page 44.
As already noted in section 3.2, the *webcrawler* provides two modes of execution: one tailored for indexing newly added or modified resources and one that visits previously indexed *place* and *service pages* and is intended to be used for regular crawling. Both modes are explained more thoroughly in this section.

The class hierarchy of the *webcrawler*, which can be seen in figure 4.2, is subdivided into a set of model classes, database connector classes, utility classes and classes that work tightly together with the application's main class, the `Crawler`.

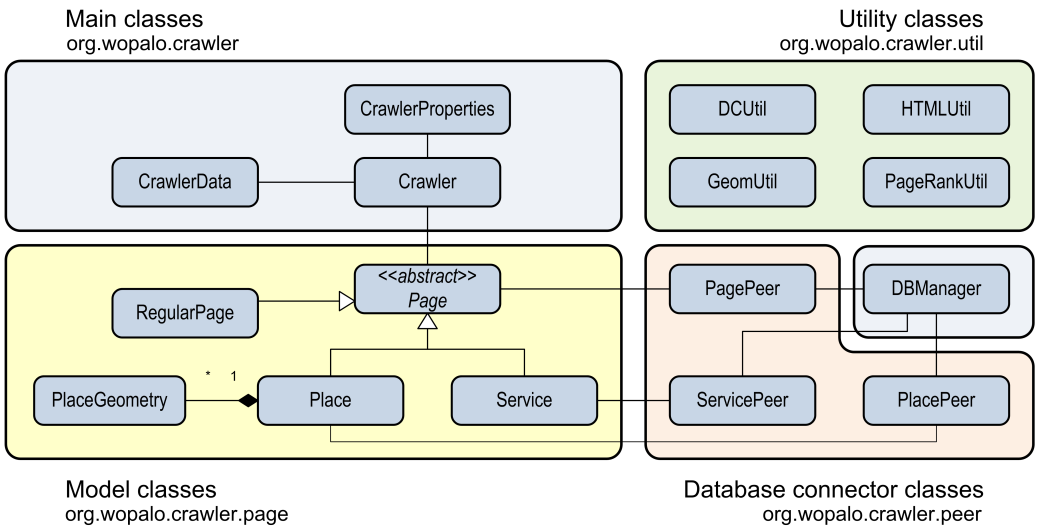


Figure 4.2: Class diagram of the *webcrawler*

The model classes are used to model web pages and can either represent a regular web page not containing any valuable information with respect to *location-based services*, a *place page* or a *service page*. They store relevant information about the respective web resource such as hyperlinks, for instance. Each of the model classes references its corresponding peer class, which is responsible for handling database specific operations like storing the page or retrieving it.

Since the model- and database-related classes do not contain specific implementation details, only the classes working together with the `Crawler` and the helper and utility classes are discussed in the following sections. The crawl modes are discussed with the `Crawler` class.

4.2.1 Main Classes

`CrawlerData`

The `CrawlerData` class keeps track of all crawler relevant data. It stores a list of URLs to web pages not yet crawled and provides methods to manipulate this list. Web resources already processed will not be added to it to avoid infinite recursion.
The `CrawlerData` class additionally provides a method of checking whether the *webcrawler* is permitted to crawl a specific resource. This can be determined by parsing the robots.txt file of the host contained in the host's web root directory. The robots.txt defines disallow-paths, instructions for webcrawlers about paths and files that should be omitted when crawling web pages from that host; this is called the *Robots Exclusion Protocol*. Unfortunately, there is no official standard or RFC[9] for the *Robots Exclusion Protocol* but all well-known webcrawlers follow the instructions defined in the robots.txt file. A sample robots.txt file can be seen in the listing below.

```
# robots.txt for http://www.example.com/

# disallow-paths for all agents
User-agent: *
Disallow: /cyberworld/map/
Disallow: /tmp/
Disallow: /foo.html

# no disallows for samplebot
User-agent: samplebot
Disallow:
```

Listing 4.1: Sample content of http://www.example.com/robots.txt

If a host does not have a robots.txt file contained in its root directory, a webcrawler is basically allowed to crawl all hosted pages. If the file is present and contains a user-agent entry matching the name of a webcrawler, the disallow-paths defined in the respective section are to be used by that crawler (e.g., according to listing 4.1 samplebot is allowed to crawl all web resources from http://www.example.com). Otherwise, the disallow-paths defined for all webcrawlers are to be used, matching the "User-agent: *" entry.
The WoPaLo *webcrawler*, named *wopaloC*, conforms to the *Robots Exclusion Protocol* and proceeds, as outlined above, to decide if a specific web resource can be crawled. For each host the robots.txt file is processed once and the disallow-paths are stored and reused by the `CrawlerData` class.

`Crawler`

The `Crawler` is the main class of the application and contains the main method. Upon invocation of the main method it checks whether the `CrawlerProperties` are valid and tests if an argument has been provided with the command line. It creates an instance of the `Crawler`, initiates the database connection using the `DBManager` class and starts a *regular crawl* if no argument was provided and a *single resource crawl* otherwise. Crawling is terminated if all resources are processed.
Independent of the crawl mode, the `Crawler` visits the web resources in a breadth-first fashion. Links found on crawled web pages are added to a list of not yet processed resources

[9] The RFC draft can be found at: *http://www.robotstxt.org/norobots-rfc.txt*

and are crawled in turn. The reason for traversing web resources in this manner is to avoid stack overflows due to deep recursions. However, sometimes it is important to process certain links right away. This is the case for *service pages* and *place pages* that have META links which reference other *place pages* (e.g., a *service page* referencing a *place page* or a *place page* referencing its parent *place page*). The META links are processed right away to check whether the referenced resource represents a valid *place page*, before modelling this reference in the database. Moreover, to model this reference it is required to have both the referencing and the referenced web resource stored in the database, which is another reason to follow the reference first.

The crawl modes are discussed in more detail below.

Regular Crawl The `Crawler` retrieves a list of previously indexed web resources from the `CrawlerData` class and starts crawling them in turn. The `Crawler` sleeps for a predefined amount of time, before actually processing a resource from the list to limit the number of requests per minute for a host. It checks whether the resource has already been processed in this run or if the resource has been crawled within a certain interval. If any of these criteria is true the web resource will not be crawled in this iteration.
The `Crawler` checks if it is allowed to process the resource conforming to the *Robots Exclusion Protocol* and crawls the web page with the help of the `HTMLUtil` class. It determines the *Google PageRank* of the resource, retrieves the `MetaTags` and `MetaLinks` and creates the appropriate `Page` instance. If this instance represents a regular web page it adds the contained HTML links to the list of web resources not yet crawled and starts processing the next web resource.
If the web resource is either a *place* or *service page*, the `Crawler` resolves existing dependencies (e.g., a *service page* linking a *place page*) by crawling these web resources right away. It then stores the resource currently being crawled using the corresponding peer object. The HTML links are added to the list of resources to process and the next resource is crawled.

Single Resource Crawl The `Crawler` checks whether the provided command line argument represents a valid URL. It sleeps for a predefined amount of time and checks if the resource has already been processed and if the resource can be processed based on the *Robots Exclusion Protocol*. It then crawls the page, extracts the available `MetaTags` and `MetaLinks` and creates the appropriate `Page` instance. If the `Page` has not changed with respect to the *place* or *service* relevant metadata, crawling of this web resource is terminated.
If the `Page` is a regular web page, the `Crawler` adds the HTML links found on the web page to the list of web resources to process and considers the next resource. The HTML links will only be followed for a certain depth if not referencing any *place* or *service pages*. If the `Page` represents a *place page* or a *service page*, the `Crawler` resolves existing dependencies by immediately crawling the respective web resources. When the required metadata is present and valid, the `Page` is stored in the database.
The HTML links are added to the list of resources to process and the next resource is crawled.

4.2.2 Helper Classes

`DCUtil`

The `DCUtil` class can be used to determine whether the type of `MetaTag` or `MetaLink` encountered on a web page represents a DC element one is actually interested in (e.g., name, description) and the type of the element. The DCMI proposes a method of embedding elements of the DC vocabulary using META tags on a webpage. However, the *webcrawler* is tolerant if a web page does not specify certain things (e.g, the absence of the DC- and DCTERMS-schema definitions).

`HTMLUtil`

The `HTMLUtil` class is used for parsing web pages. It provides methods for retrieving a web page and its `MetaTag`s and `MetaLink`s. When downloading a web page using the method provided by the `HTMLUtil` class, a request-header is send with the property "user-agent" set to the name of the *webcrawler*, called *wopaloC*. This is done so that the host is aware of the *webcrawler* it is visited by.

`GeomUtil`

This utility-class is used for parsing strings encoded in the location format described in section 3.1.4 on page 19.

`PageRankUtil`

This utility-class can be used to determine the *Google PageRank* of a web page. This metric is used both for *place* and *service*-ranking as described in sections 3.3.4 and 3.3.5.

`CrawlerProperties`

The `CrawlerProperties` class contains all constants used by the *webcrawler*. The constants are defined in a configuration file provided by the system and may be overridden by a user configuration file; see appendix B.1.

4.3 Search Engine

The *search engine* enables a user to query for a ranked set of *places* in the vicinity of a geospatial position and for a ranked set of *services* associated with a *place*. The *search engine* has been developed using RoR and utilizes the same database as the *webcrawler* (for a schematic database view, see figure 4.3).

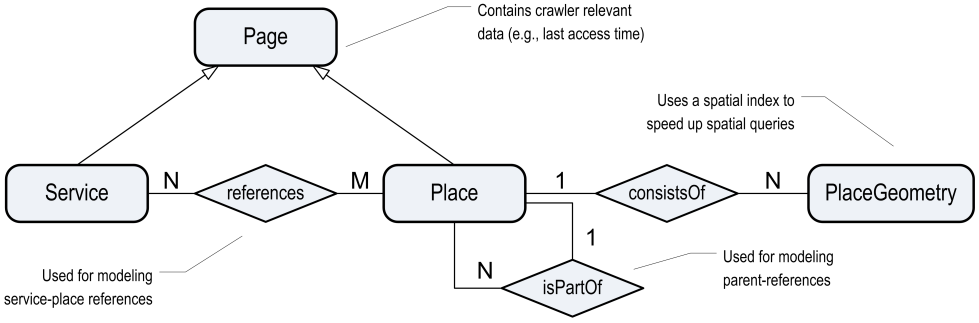


Figure 4.3: Schematic overview of the database used both by the *webcrawler* and the *search engine*

The *search engine* can return search results in various formats. These formats and the interesting rails routes are described below. The possible input parameters for the shown routes are covered in appendix C.1.

4.3.1 Available Formats

Currently there are only two formats available, namely:

HTML This is the default format that will also be used if the format argument is not set for a route.

compact The result returned when using this format is of MIME type text/plain and is returned in a representation that is both compact and simply parsable. This format is used by the *mobile phone client*, which does not need a HTML representation of the query result as it renders the result using the mobile phone's default widgets.

The text format returned for spatial and for service queries is explained with the individual routes in appendix C.1.

4.3.2 Routes

GET `/places(.:format)`

This route is used for spatial queries and requires at least a position parameter to be passed along as an argument with the request. The position consists of a latitude/longitude pair encoded using WGS84 and an optional level coordinate. An example request can be seen in the listing below.

```
GET /places.compact?p=47.3802847176866,8.54478836059571&count=8&from=0
```

The passed position defines the *point of interest* and is used as the center of the *search mask*. The *search mask* constrains the size of the neighborhood considered when a query on the

database is performed and has dimensions about $200m$*$200m$. It is used to limit the amount of considered *places* to a handful of *places* useful to the user (e.g., a user will probably not benefit much from *services* bound to a *place* at a distance of 1km when on foot). Optionally it is possible to provide the lower left and upper right position (encoded using WGS84) of a custom *search mask* as an argument, which will replace the default mask.
The *search mask* is then used to construct the SQL query for retrieving all *places* spatially intersecting with it. More precisely, the query will return all *places* having a bounding box which intersects spatially with the *search mask*. The advantage of using bounding boxes is that the query can utilize the spatial index defined for the geometric objects in the database, resulting in a tremendous speedup.
The *places* returned by the SQL query are filtered. If the geographic position provided represents an *indoor position*, the outdoor *places* are filtered out; if the position represents an *outdoor position*, indoor *places* are filtered out.
Each of the remaining *places* is then ranked according to the criteria discussed in section 3.3, depending on the geometric properties of the *place* and its location compared to the point of interest (e.g., minimum object distance). Some of these properties could be computed using the capabilities of the PostGIS extension for PostgreSQL, but since neither the maximum object distance nor the distance to a polygons centroid can be computed by the PostGIS extension, the properties are computed lazily by the *search engine* when being accessed for the first time.
The *places* are sorted according to their rank and the list of returned *places* is truncated if a certain threshold is reached. The remaining *places* are returned in the requested format; the *places* returned for the sample request can be seen below.[10]

```
25,1,8
ETHZ IFW,ETH Informatikgebäude,http://places.wopalo.org/places/17
IFW Dachterasse,Dachterasse IFW Gebäude ETH Zürich,http://places.wopalo.org/places/132
RZ-Building,RZ-Building of ETH Zurich,http://places.wopalo.org/places/32
Gangway between IFW and RZ,Connection between IFW and RZ buildings,http://places.wopalo.org/places/33
Test Building,Test of an indoor building,http://places.wopalo.org/places/24
Crosswalk,Crosswalk,http://places.wopalo.org/places/135
Church of Liebfrauen,http://de.wikipedia.org/wiki/Liebfrauenkirche_...,http://places.wopalo.org/places/28
Haldenegg,Trambahnhaltestelle Haldenegg,http://places.wopalo.org/places/22
```

GET `/services(.:format)`

This route can be used to query for *service pages* referencing a *place page* with the given URL. A sample request can be seen below.

```
GET /services.compact?url=http://places.wopalo.org/places/22
```

If a *place* with the provided URL is contained in the database, an SQL query is performed retrieving all *services* referencing this *place*. The *services* are ranked according to the criteria described in section 3.3.5 and sorted after their rank values. The list of ranked *services* is truncated to limit the size of the result and can be further adjusted by providing parameters for result mask (see appendix C.1 for the API description).
The *services* are rendered in the requested format; the *services* returned for the sample request can be seen in the listing below.

```
1,1,1
VBZ Fahrplaninfo,Anschlüsse ab Zürich Haldenegg,http://vbzservice.wopalo.org/station/299915
```

[10] Some of the URLs have been replaced with "..." for the sake of readability.

GET `/hint`

This route can be used to inform the *search engine* about a *place* or *service page* which might have been updated and should be re-indexed by the *webcrawler*. A sample request can be seen below.

```
GET /hint?url=http://places.wopalo.org/places/3
```

The route requires the URL of the web page as an argument and creates an instance of the *webcrawler*, passing the URL as an input argument to it. The crawler proceeds as outlined in section 3.2 and described more thoroughly in section 4.2.

4.4 Mobile Phone Client

The *mobile phone client* has been developed to allow the user to consume *location-based services* when being in-situ. It has been implemented for the Nokia N95 8GB using Python and uses the mobile phone's default widgets to fit with its look and feel.

The sections below describe the main classes and the existing utility classes of the *mobile phone client*. The class diagram of the *mobile phone client* can be seen in figure 4.4. Since all user interactions are handled by the `WoPaLoClient` class, the possible user interactions are described there.

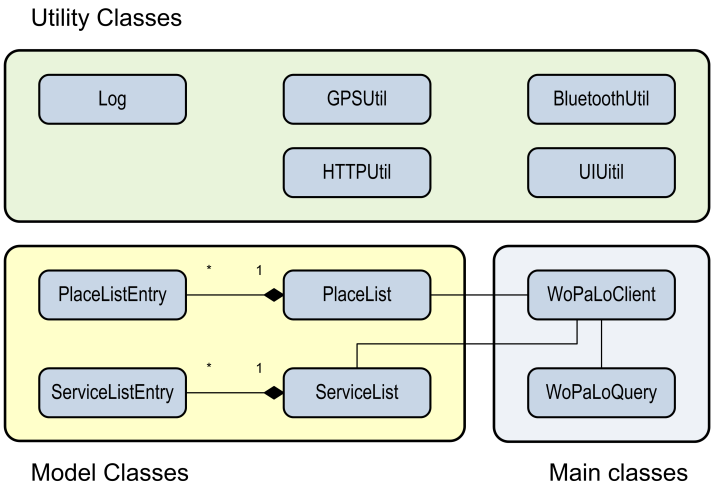


Figure 4.4: Class diagram of the *mobile phone client*

4.4.1 Main Classes

`WoPaLoClient`

This is the application's main class which is responsible for handling user input and executing the appropriate actions.
When run for the first time, the `WoPaLoClient` displays the main menu. Currently the main menu allows the user to select from two possible actions: display a list of outdoor *places* or display a list of indoor *places*. A screenshot displaying the applications main menu can be seen in figure 4.5(a).

When a user requests to view outdoor *places*, the `GPSUtil` class is asked to perform a GPS query and return the spatial position. If this query is not successful, it is assumed that the mobile device is located indoors. Bluetooth devices in the vicinity are then sensed is by the `BluetoothUtil` class which will try to determine the spatial position as well (see the `BluetoothUtil` class for more details on how this works). If the spatial position can not be determined the application's main menu is re-rendered. Otherwise the `WoPaLoQuery` class is used to retrieve a list of *places* for the spatial position.

(a) Main Menu (b) Places

Figure 4.5: Main views of the *mobile phone client*

The returned list is extended by "Next results" buttons and "Previous results" buttons by the `WoPaLoClient` depending on the number of results returned by the query and is displayed on the screen (see figure 4.5(b)). For each of the returned *places* a context-sensitive menu can be displayed, which allows the user to go back to the main menu, show the *place page* of the selected *place* using the internal web browser, query for the associated *services* or quit the application. A screenshot showing the context-sensitive menu displayed for places can be seen in figure 4.6(a).
If the user chooses to view the *services* associated with a *place* by either clicking on the *place* or selecting the respective entry in the context-sensitive menu, the `WoPaLoClient` uses the `WoPaLoQuery` class to query the *search engine*. Again, "Next results"- and "Previous results"-buttons are added to the list depending on the number of returned results and the list is displayed on the screen. When the user clicks on a *service*, the `WoPaLoClient` opens the external browser and dispays the corresponding *service page*, thus enabling the user to actually consume a *location-based service*.
The context-sensitive menu displayed for a *service* lets the user decide to display the main menu of the application or to go "Back" (see figure 4.6(b)). As the `WoPaLoClient` keeps track of recently retrieved *places*, selecting "Back" will re-render the previously displayed *places* without querying the *search engine* again.

When the user chooses to view indoor *places*, the `WoPaLoClient` will only use the `BlueToothUtil` class for performing spatial positioning. If a list of *places* is returned, the rendered menus and the behavior of the application is the same as when querying for outdoor *places*, hence the same actions as described above are available to the user.

(a) Places (b) Services

Figure 4.6: Context-sensitive menus of the *mobile phone client*

`WoPaLoQuery`

This class is responsible for performing *search engine* queries. Depending on the query type (e.g., a *place* query or a *service* query), it assembles the appropriate query string and queries the *search engine* using the `HTTPUtil` class. It requests the *search engine* to return the query results in the compact format (see section 4.3.1) as it uses the mobile phone's default widgets to render the results.

4.4.2 Utility Classes

`HTTPUtil`

This utility class can be used to open the internal web browser for a given URL or to retrieve the content from a web page. The quirk with the latter is that the content of a web page can not be read directly due to limitations of PyS60. The web page is downloaded and stored in a temporary file first and then read from there.

`GPSUtil`

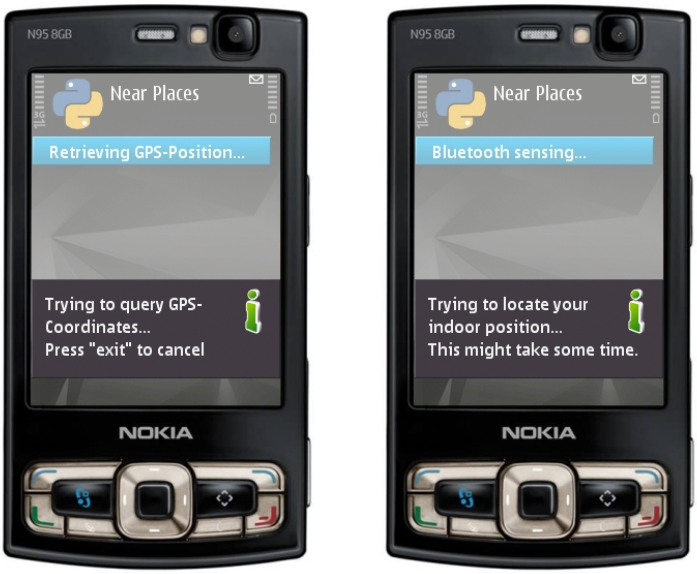
The `GPSUtil` class can be used to determine the spatial position by using GPS. It is used to limit a GPS inquiry in time, as there are cases when no GPS positioning is possible (e.g., being indoor) and as a user should not have to wait longer than a certain amount of time. The inquiry is automatically terminated after reaching the timeout. Additionally, the user may terminate the GPS inquiry if he decides to.
The `GPSUtil` class informs the user of any ongoing GPS inquiry and offers the option to abort the inquiry. A screenshot showing an ongoing inquiry can be seen in figure 4.7(a).

`BluetoothUtil`

The `BluetoothUtil` class can be used to determine the spatial position by Bluetooth sensing. The method for identifying the position is quite simple: it tries to match the MAC address of a sensed device with a list of known MAC addresses. Each of these MAC addresses has an associated geospatial position, and therefore the entries in the list are encoded as tuples of (MAC address, geospatial position). The positions represent *indoor positions* (latitude/longitude/level) as it is assumed that Bluetooth devices are encountered indoors. The list is stored in a file on the device which is processed when creating an instance of the `BluetoothUtil` class. An example mapping file can be seen in the listing below.

```
00:1F:3A:FC:05:84 47.3802847176866,8.544788360595703,0
00:1F:3A:FC:05:84 47.3802847176866,8.544788360595703,0
```

Just like the `GPSUtil` class, the time used for spatial positioning is limited. Bluetooth sensing is terminated when the timeout is reached or if the user opts to terminate it.
The `BluetoothUtil` class informs the user of ongoing Bluetooth sensing and the option to abort it. For a screenshot showing ongoing Bluetooth sensing see figure 4.7(b).



(a) GPS inquiry (b) Bluetooth sensing

Figure 4.7: Notification windows of the *mobile phone client*

`UIUtil`

This utility class provides methods for displaying a dialog and temporarily setting the on-screen menu. It is used to inform the user about ongoing spatial positioning and is used by the `GPSUtil` and `BluetoothUtil` classes.

4.5 Frontend

The main purpose of the *frontend* is to simplify the definition of *places* for the user. It enables the user to define both outdoor and indoor *places* on a map displayed in the birds-eye perspective. Additionally, it offers the possibility to store the created *places* as the user does not typically want to host the associated *place pages*. The *frontend* was developed as a web application using RoR at the sever backend and JavaScript together with the Google Maps JavaScript API at the client side to visualize defined *places*.

Since outdoor and indoor *places* are defined differently they are covered in separate sections.

4.5.1 Defining Outdoor Places

Outdoor *places* can be created and modified on top of a scrollable and zoomable Google Map displaying satellite images of the Earth and can be shown in fullscreen. To aid the user in defining new *places* a context-sensitive help is always displayed in the lower right-hand corner of the map. An image showing the outdoor view can be seen in figure 4.8.

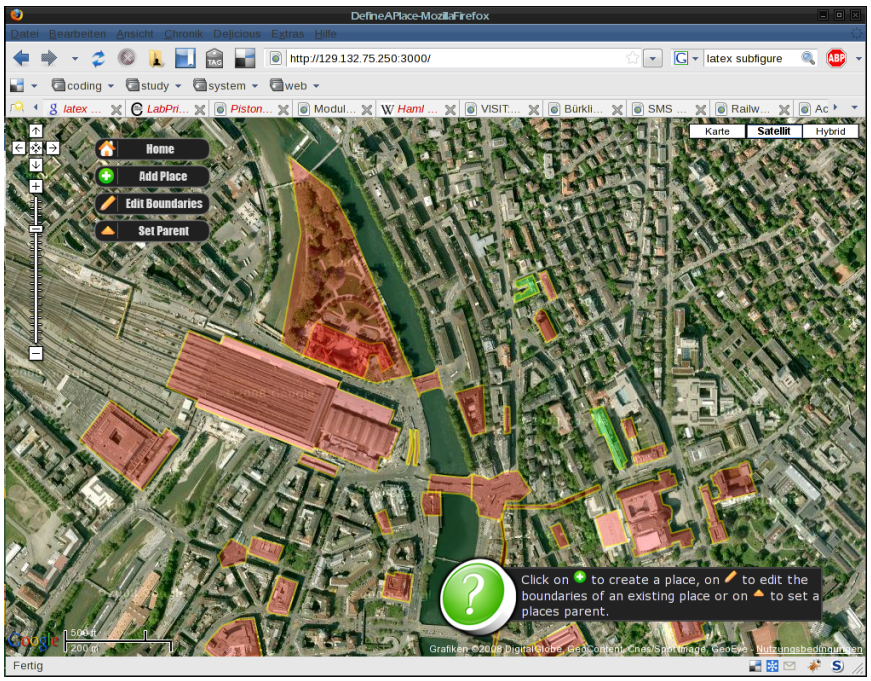


Figure 4.8: Outdoor view of the *frontend*

Interactions with the map are handled on the client side, which is why a great portion of the application logic is written in JavaScript (see section 4.5.1.1). Communication with the server is implemented with AJAX requests[11] to the appropriate routes, discussed in section 4.5.1.2. The responses are processed at the client side and depend on the returned HTTPResponse status.

[11] Using the Prototype library which works with all common browsers and is included by default with RoR.

4.5.1.1 JavaScript Classes

As already indicated, the JavaScript classes handle interactions with the map. The classes of interest and the application level methods of the main JavaScript class are discussed below. An overview of the implemented JavaScript classes can be seen in figure 4.9.

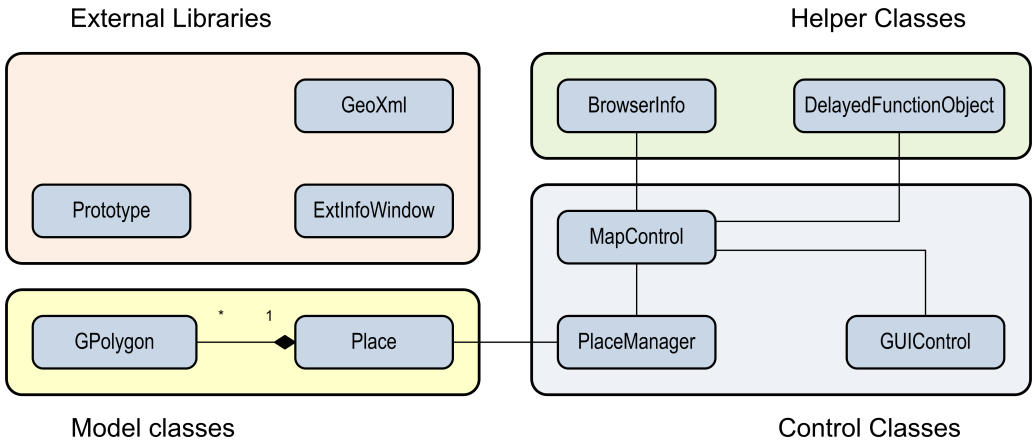


Figure 4.9: JavaScript class hierarchy

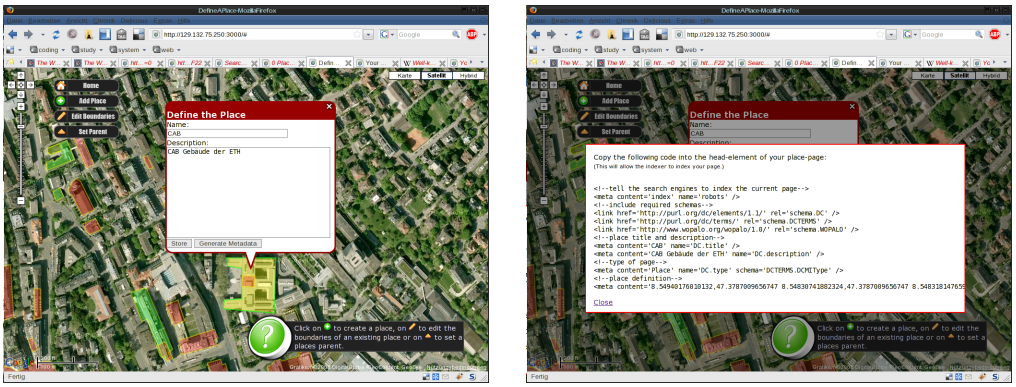
`MapControl`

This is the main JavaScript class responsible for controlling objects on the map, user interactions with these objects by registered event handlers, and handling communication with the server backend. It contains a set of methods which are triggered by different user actions. These actions are:

update map This action is triggered when moving or zooming the map. It removes previously visible *places* that go out of sight and renders other *places* as they become visible. Compared to redrawing all *places* contained in the current viewport this approach has the benefit of reducing the amount of data required to be transmitted by the server. The client sends a request to the server retrieving all *places* in KML format that are not yet contained in the current viewport. These *places* are then parsed locally by the client using the GeoXml library and are rendered onto the map. The GeoXml KML parser is used instead of the default KML parser provided with the Google Maps JavaScript, as the latter tends to do certain optimizations expressing itself in *places* not being removed correctly from the map.
Updating the map is an expensive and resource consuming operation with respect to the amount of server requests to the *frontend* and the time required to parse the returned *places*. As zooming the map by more than one zoom level will triggers an event for every zoom level, *places* contained in the current viewport will be re-rendered several times within a second. The same applies to the scenario where a user moves the map from point A to point B by dragging it incrementally by a short distance. Actually, it

is not necessary to display the *places* while a user is still moving or zooming the map, since the user would not be interacting with the map if interested in *places* contained in the currently displayed area. Therefore all server requests regarding *place* updates are cancelled, while a user is still interacting with the map. A single request is sent to the *frontend* after the user has stopped interacting with the map for half a second. The `DelayedFunctionObject` was used to implement the desired behavior.

create place The user can create a polygon by defining vertexes when clicking on the map. If the polygon is closed by either clicking on the last or the first vertex, another polygon can be added to the current *place* in the same manner or the user may select to finish the creation of the *place* by clicking the "Home" button. This sends a request to the server which returns a dialog where the user may type in a name and description for the *place*. At this point the user can store the *place* (see image 4.10(a)) or alternatively request the display of an on-screen window containing metadata representing the defined *place* (see image 4.10(b)) which can be copied into a HTML page.



(a) Store *place* (b) Generate *place* metadata

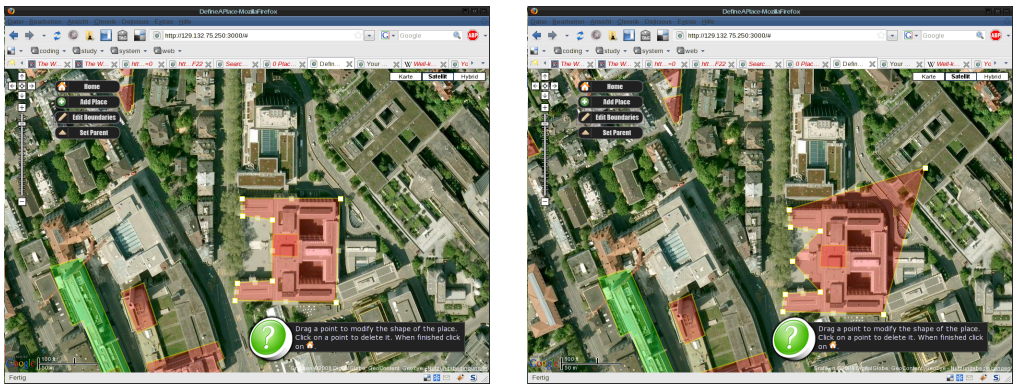
Figure 4.10: Create a place

When a newly defined *place* has not yet been stored, clicking on it will always bring up the save dialog. Therefore accidentally closing the save dialog does not have any consequences.

modify place The `MapControl` class displays vertexes for each point of the selected polygon. Between each two neighboring vertexes a ghost vertex is rendered (see image 4.11(a)). The vertexes can be dragged around by the user; dragging a ghost vertex results in a new vertex for the polygon (see image 4.11(b)).

When the user clicks the "Home" button the server returns a dialog allowing the user to confirm the changes.

set parent The user can define a *place*'s parent by first selecting the *place* and clicking on the parent *place* afterwards. If the user has finished and clicks the "Home" button, the server returns a dialog to save the changes.

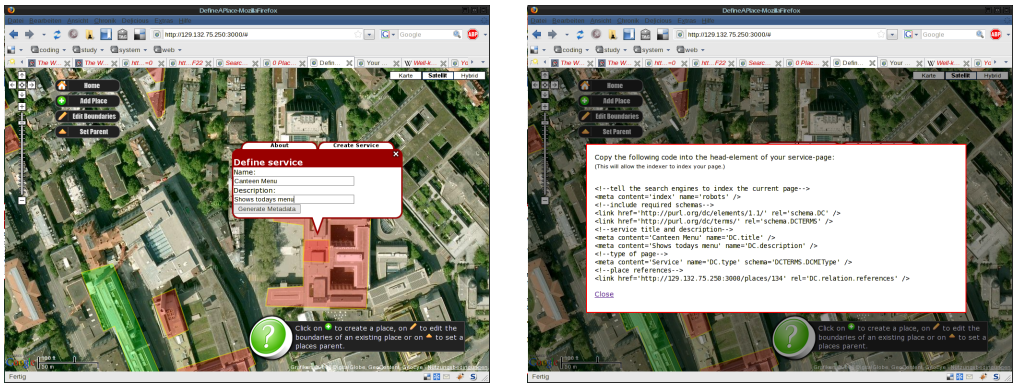


(a) Edit *place* (b) Edited *place*

Figure 4.11: Modify the *place* boundaries by dragging normal or ghost vertexes

`PlaceManager`

The `PlaceManager` is used to manage all *places* currently rendered on the map. It remembers the currently selected *place* and polygon and provides methods to set the behavior of managed *places* when clicking on them.
The default behavior when clicking on a *place* on the map is to show an information dialog for it. Besides showing the summary for the *place*, this dialog enables the user to generate metadata for a *service page* that should reference the *place page* of the selected *place* (see image 4.12(a)). The metadata is displayed using an on-screen window and can be copied into the definition of a HTML page (see image 4.12(b)).



(a) Define *service* (b) Generated metadata

Figure 4.12: Create a *service* for a *place*

`GUIControl`

The `GUIControl` class is used to create and manage the GUI buttons.

4.5.1.2 Routes

Since most of the server side routes return HTML code for dialogs to be rendered on the map, only the interesting ones are listed below.

`GET /map/viewport(.:format)`

This route is used to query for *places* intersecting (contained or overlapping) with the current viewport of the map and hosted by the *frontend*. Besides the viewport parameters consisting of a latitude/longitude pair for the lower-left and one for the upper-right corner of the viewport, it additionally accepts a semicolon separated list of *place* ids which determines the *places* that will be removed from the result set. This can be used to avoid the retrieval of *places* already displayed on the map. An example request can be seen in the listing below.

```
GET /map/viewport.kml?ne=47.39;8.58&sw=47.36;8.50&blacklist=1;81;83;23;41
```

Since there may be a lot of *places* intersecting with the current viewport, the *places* are filtered according to their size. *Places* filling the whole viewport or *places* hardly visible will not be returned. The reason for this is to limit the returned results to an appropriate subset (e.g., it makes no sense to display the *place* representing Switzerland if the map is zoomed in on Zurich, Central).
The server can return the collection of *places* in two different formats: JSON and KML. JSON is a human readable format for representing simple data structures that can be easily processed and used with JavaScript. KML is an XML based language and is the location-encoding standard preferred by Google Maps. Besides geographic data, KML may contain rendering specific information such as border- and fill-colors of polygons or iconstyles for markers.

`GET /places/:id`

This route is used for the *place page* of a *place* hosted by the *frontend* and will typically be crawled by a *webcrawler*. The metadata of the *place* is encoded as described in section 3.1.

4.5.2 Defining Indoor Places

The *frontend* enables a user to create two types of indoor *places*. An indoor *place* may either be a floor or a room located on a previously defined floor.
It would have been desirable to render indoor *places* using the same map as for outdoor *places* and to provide a button to switch from the outdoor view to the indoor view and vice versa (e.g., select a building and click the button to switch to the indoor view only displaying indoor *places*). Besides that, the use of a scroll bar capable of switching between the displayed floors and contained rooms would have been convenient. However, due to time constraints indoor *places* have been created as follows.

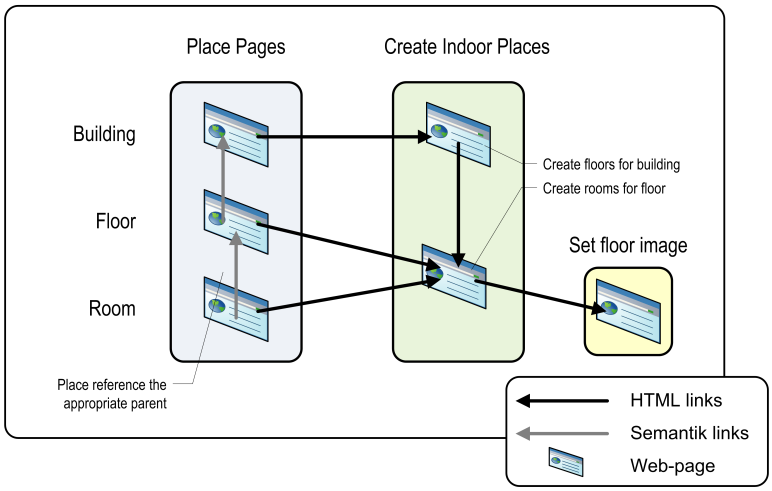


Figure 4.13: Creating indoor *places*

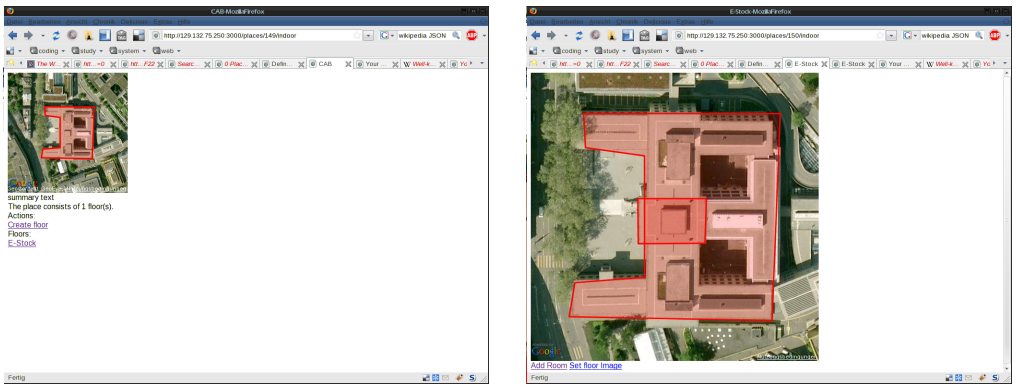
Each *place page* provides a HTML link to a separate web page which can then be used to define indoor *places* (see figure 4.13). Depending on the type of *place page* (e.g., a floor) the linked web page will render different content.

The types distinguished are:

Building The returned web page displays HTML links to all floors contained within a given building and lets the user define new floors. In order to define a floor, its name, description and level need to be provided. An image showing the HTML page returned for a building can be seen in figure 4.14(a).
Newly created floors have the same boundaries as the building.

Floor The returned web page contains a Google Map showing the floor and the rooms belonging to this floor. A user may add new rooms to the floor in a point-and-click like manner using the displayed map. When the user closes the polygon defining the spatial whereabouts of the *place* he may specify its name and description and store it on the server. Figure 4.14(b) shows the HTML page returned for a floor.
It is possible for a user to upload a floor image for the current floor, which will be displayed below the *places* on the map; see section 4.5.2.1 for details.

Room For a room the same content is returned as for the floor the room is in.



(a) Create floors for the building (b) Create rooms for the floor

Figure 4.14: Create indoor *places*

4.5.2.1 Floor Image

There is a dedicated web page for each floor which can be used to upload the floor image. If the image has been uploaded, the same web page allows the user to position the image to fit his or her needs. To position the image two maps are displayed, one showing the image in its original size (right region of image 4.15(a)) and one showing the floor polygon and a scaled down version of the uploaded image (left region of image 4.15(a)). Both maps have draggable reference markers which can be used to position the image (see image 4.15(b)). The map showing both the floor and the scaled image is updated each time a marker is dragged to reflect the changes made. Since it is not possible to rotate an image using the Google Maps API, this is done by the *frontend* which can return arbitrarily rotated and scaled images with the help of RMagick[12]. When the user is satisfied with the position, he or she can save the changes made and, by doing so, he or she is automatically redirected to the indoor view of the floor.

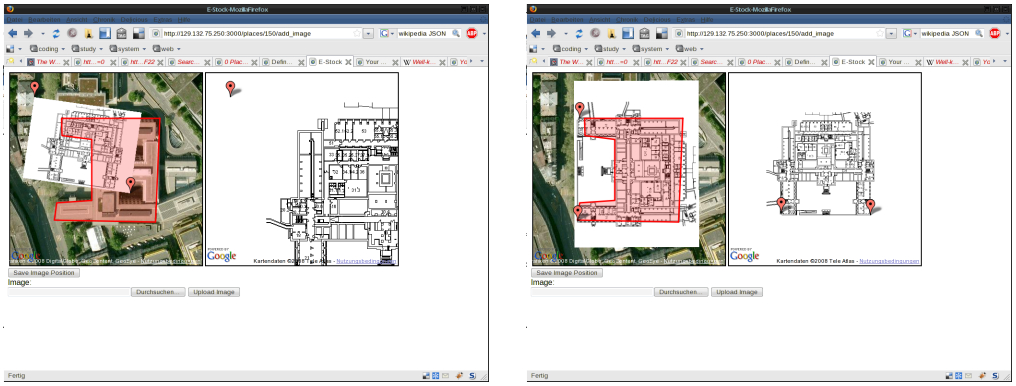
4.5.2.2 Routes

The following routes are the most important when working with the indoor view of a *place*.

`GET /place/:id/indoor`

This renders a web page enabling the user to define indoor *places* routed at the *place* with the given id. If the *place* is either a floor or a room, the returned web page contains a map displaying the floor and the rooms contained on the floor. Otherwise the web page displays a list of floors contained in the building and lets the user add floors.

[12] Ruby ImageMagick bindings.



(a) Begin of image positioning

(b) Finished image positioning

Figure 4.15: Setting a floor image

`GET /place/:id/image`

This route is used for returning the image belonging to a *place*. Images can only be uploaded for floors and are stored in the database.
The route accepts paremeters that can be used to return a scaled and/or rotated image. An example request is shown in the listing below.

```
GET /places/18/image?rot=338.5946917230823&scale=0.27128718187291617
```

4.6 VBZ Service

The *VBZ service* has been implemented in order to have a useful showcase of a *service*. It has been developed as a separate web application and provides *service pages* for tram stations located in Zurich. These pages display electronic timetables for trams departing at certain stations and reference the corresponding *place pages*. The *service pages* will be crawled by the *webcrawler* in the same manner as any other web page.

The timetable data for the *VBZ service* has been kindly provided by the VBZ. The VBZ raises no claims about the accuracy of the supplied data.

The database layout can be seen in Appendix A.1, the available formats and the interesting rails routes are described below in more detail.

4.6.1 Available Formats

There are two available formats:

HTML The default format which will be used if no format is specified. It renders a HTML page designed for access by a generic client.

nokiaN95 A format also returning content of MIME type text/html but adapted for the screen size and font types supported by the Nokia N95 8GB.

4.6.2 Routes

GET `/station/:city/:station(.:format)`
GET `/station/:id(.:format)`

These routes are used to display timetable data for certain stations. Depending on the route used, the corresponding stations are retrieved from the database. A station stored in the database has a unique id but no unique name. The reason for that is that there are places in Zurich where trams depart in more than two ways (e.g., Central) and the respective stations share the same name. Therefore the route taking an id as parameter has to be used if the timetable data for the specific station is requested. The route accepting city and station names as parameters returns timetable data for stations with the same name.

The example below shows two requests returning the same result.

```
GET /station/101278
GET /station/Zürich/Hölderlinstrasse
```

If the name or id entered are known to the system, a database query is performed which retrieves the chronologically next 10 tram departures for the respective station(s). These departures are dynamically rendered using a HTML page.
As the route taking an id as parameter uniquely identifies a station, these HTML pages represent the *service pages* of the *VBZ service*. Their metadata is set as defined in section 3.1.3, but only if the *service pages* actually reference a *place page*. This is the case if the URL of a *place page* is associated with a station in the database[13]. The *place pages* for the respective *places* have been created using the *frontend*.

Figure 4.16 shows the *service page* for "Zürich, Bahnhofsquai", rendered by the *mobile phone client*.

[13] These entries have been set manually using a database management tool.

Figure 4.16: *Service page* for "Zürich, Bahnhofsquai"

GET `/bootstrap`

This route can only be called locally and will initialize the database. This is done by parsing station, tram, operating days and stops data available in the HAFAS raw format[14] with respect to their dependencies (e.g., parsing station data before stops data). When processing one of these files, all SQL statements are encapsulated in a single transaction to accelerate the process as a file may hold up to 5MB of raw data.

GET `/referenced`

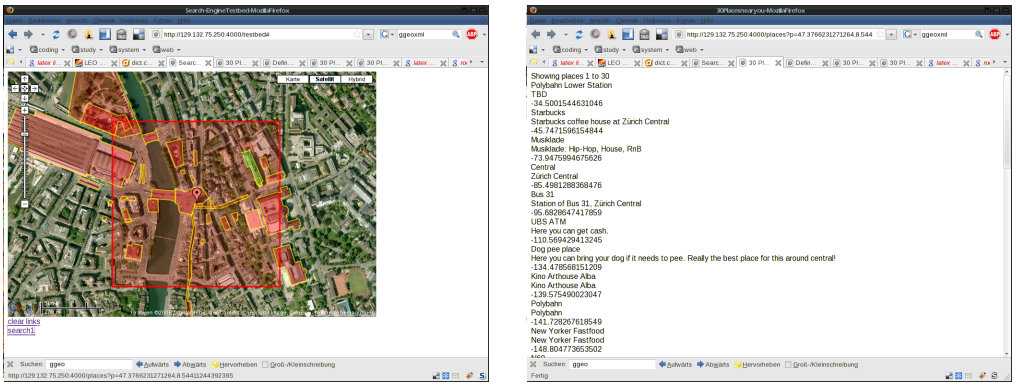
A request to this route retrieves all stations having *service pages* that reference *place pages* and returns a web page displaying a collection of HTML links to these *service pages*.

The purpose of this route is to provide a start page for the *webcrawler*.

[14]The HAFAS format is used to exchange timetable data between different information retrieval systems.

4.7 Testbed

The *testbed* was designed to determine the constants and weights for the individual penalty functions used by the *search engine* when performing *spatial queries*.
The *testbed* was realized as a HTML page and displays a Google Map. By clicking on the map it dynamically generates a HTML link and appends it to the page (see image 4.17(a)). The HTML link represents a *search engine* query using the spatial position clicked on as the required position parameter. By clicking on the link, the *search engine* will do the *spatial query* and present the ranked results (see image 4.17(b)).



(a) Testing a spatial query

(b) Results returned for the query

Figure 4.17: Evaluation of *spatial queries* using the *testbed*

To be able to compare the returned results of the query with the ones expected, the *places* in the vicinity of the clicked spatial position are rendered onto the map together with a box having the same dimensions as the *search mask* of the *search engine* (see image 4.17(a)). The *places* displayed are the ones defined when using the *frontend* and are retrieved by using the viewport route.

The constants for the spatial penalty functions were determined by evaluating the spatial ranking results using various scenarios, e.g. near and large *place* vs. small and far away *place*. The constants were incrementally adjusted to return results intuitive for a user when considering the ranking assumptions: the smaller and nearer a *place* to a queried spatial position the better (see section 3.3.1).

Figure 4.18 shows one possible ranking scenario and displays two *places*: a gray one and a black one. For spatial positions located in a certain area it is intuitively clear which *places* should be ranked better. For the gray and the black *place* in the figure these areas are displayed in yellow and orange colors respectively. Outside of this area however, it is difficult to tell which *place* should be ranked better, even if arguing based on the assumptions. For this reason the constants were optimized for spatial positions inside of the denoted areas.

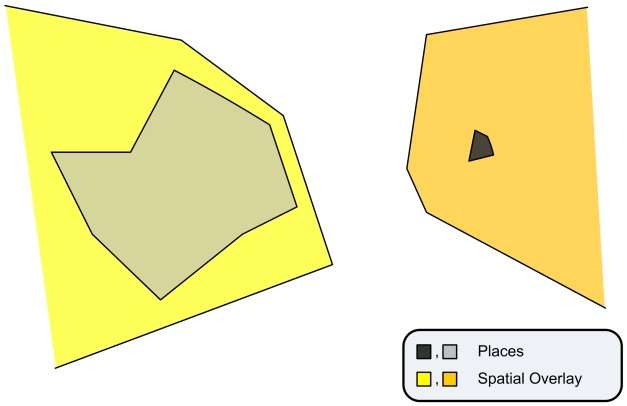


Figure 4.18: Evaluation Scenario

The constants which were determined and are actually used are the ones depicted besides the individual penalty functions described in section 3.3.3 on page 24 and following. The weights used for weighting the penalty functions are set to 1.

5 Evaluation

This chapter is devoted to a preliminary evaluation of the results returned by the *search engine* when doing *spatial queries*. To evaluate these queries, about 140 new *places* were defined in the region of Zurich. However, it is difficult to tell whether the set of used *places* corresponds to a set of *places* as they would be defined by individual users.

Please note that at the point in time when *spatial queries*, as performed by the *search engine* were evaluated, there were no other comparable search engines retrieving and ranking polygons for a geospatial position nor was there a *Web of Places and Locations*!

Spatial queries have been tested using the *testbed* that is described in section 4.7 and have been tested in the field, using the developed *mobile phone client*. The individual results are described below.

5.1 Testing With The Testbed

Spatial queries were tested for various, randomly selected spatial positions in Zurich; the *places* used in the evaluation are differed than the *places* used for training the spatial penalty functions. The ranking of the results returned for the different geospatial positions corresponds to the one expected most of the time. But when comparing the rank of two designated *places* and a position located in a geospatial area where both spatial ranking assumptions level about equally (e.g., a large *place* in the direct vicinity of the position and a smaller *place* further away), the ranking of the results might sometimes deviate from the one expected. However, this will typically not influence the ranking of other returned *places*, e.g two *places* struggling over ranks 2 and 3 will not influence other *place*'s ranks.

5.2 Testing In The Field

Spatial queries were tested in the field using the *mobile phone client* and doing spatial positioning using GPS. A position returned by GPS may deviate up to several meters and may additionally deviate due to walking around while using the *mobile phone client* and therefore passing by the returned position. However, even if the position deviates by several meters, *places* conforming to the spatial ranking assumptions (e.g., small and near) and in the near vicinity of the users position, not the one returned by GPS, will still be ranked among the first results.

6 Conclusion

This chapter concludes the work carried out with respect to the lessons learned while implementing the system, presents a small summary and outlines future work that could be done to enhance the built system.

6.1 Lessons Learned

Several of the system components were built as web applications using the Ruby on Rails framework which turned out to be a good choice. It was very convenient to work with due to the automatic separation of model, view and controller classes (MVC architecture), which enabled one to implement the application level methods before dedicating oneself to the view templates. Besides that, RoR provides facilities such as the route concept, automatic OR/M, view templates, good support by the community expressing itself in many nice plug-ins and reasonable debugging primitives. Furthermore the RoR framework and Ruby itself is pretty nice to work with due to the syntactic sugar provided in many different locations. The only drawback with the RoR framework was the utilization of a rather new version (2.0) at that point in time, which was the reason for the scarce availability of documentation, working tutorials and support.

Another application that proved to be very practical was the Batoo Rapid Prototyping Environment, which enabled fast shipment of code from computer to a mobile device without the need of using a USB cable. The Batoo Rapid Prototyping Environment was developed by Robert Adelmann (Distributed Systems Group of the ETH Zurich) and was not publicly available at the time this work was conducted.

6.2 Summary

The concept presented regarding how to represent and constitute *location-based services* in this thesis is both simple and powerful. It allows arbitrary users hosting one or more web pages to surplus the value of existing web pages by just embedding metadata into their web page. As META tags are used for enhancing a web page, its appearance stays the same. Besides, as services are represented using ordinary web pages, anything that can be modelled using a web page can represent a service - there are no limits to creativity.
Furthermore, thanks to the clean separation of the individual components and as the presented system is technically open, it is easily possible to exchange single components. One could, for example, use an alternative search engine to query for *places*.
One limitation of this system though is that it is not possible to search for specific places when doing *spatial queries*, e.g. places representing restaurants. The same applies when searching for services belonging to a place. It would certainly be useful if places and services could be

tagged to filter the uninteresting ones.

However it is difficult to say at the moment, whether the system proves itself valuable, since this depends on the quality of user-defined *services* and *places*.

6.3 Future Work

Following some ideas on how the system could be extended to enhance its quality:

Tagging

By assigning tags to places or services it would be possible for users to retrieve types of places or services in which they are interested. This could be realized by allowing individuals to tag a place or service directly with exactly one tag to minimize tag abuse. On the other hand, one could use the API of, for instance, delicious bookmarks[1] to retrieve tags for a specific web page. delicious bookmarks is available as a browser plug-in that enables users to bookmark individual web pages and assign custom tags to them. The tags can be retrieved and could be categorized by using an ontology for example.

Self-governance

As the system is open to anyone it is not secure from abuse. One could create many small places (e.g., $3 * 3m$) and position them in a way to cover larger areas, hence outruling other places when doing *spatial queries*. It would be useful to have a community-based solution to prevent abuse.

[1] *http://delicious.com/*

Bibliography

[1] Martin Bauer, Christian Becker, and Kurt Rothermel. Location models from the perspective of context-aware applications and mobile ad hoc networks. In *Personal and Ubiquitous Computing*, pages 322–328, 2002.

[2] Christian Becker and Frank Dürr. On location models for ubiquitous computing. *Personal Ubiquitous Comput.*, 9(1):20–31, 2005.

[3] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, 1998.

[4] Gregor Broll, John Hamard, Massimo Paolucci, Markus Haarländer, Matthias Wagner, Sven Siorpaes, Enrico Rukzio, Albrecht Schmidt, and Kevin Wiesner. Mobile interaction with web services through associated real world objects. In *MobileHCI '07: Proceedings of the 9th international conference on Human computer interaction with mobile devices and services*, pages 319–321, New York, NY, USA, 2007. ACM.

[5] Barry Brumitt and Steven Shafer. Topological world modeling using semantic spaces. In *In UbiComp 2001 Workshop on Location Modeling for Ubiquitous Computing*, 2001.

[6] Debbie Caswell. Uniform web presence architecture for people, places, and things. *IEEE Personal Communications*, 8:46–51, 2001.

[7] Deborah Caswell and Philippe Debaty. Creating web representations for places. In *HUC '00: Proceedings of the 2nd international symposium on Handheld and Ubiquitous Computing*, pages 114–126, London, UK, 2000. Springer-Verlag.

[8] Keith Cheverst, Nigel Davies, Keith Mitchell, Adrian Friday, and Christos Efstratiou. Developing a context-aware electronic tourist guide: Some issues and experiences. pages 17–24. ACM Press, 2000.

[9] F. Dawson and T. Howes. vcard mime directory profile, 1998.

[10] Svetlana Domnitcheva. Location modeling: State of the art and challenges. Workshop on Location Modeling for Ubiquitous Computing, 2001.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[12] Greg Janée and James Frew. Spatial search, ranking, and interoperability.

[13] Changhao Jiang and Peter Steenkiste. A hybrid location model with a computable location identifier for ubiquitous computing. In *UbiComp '02: Proceedings of the 4th international conference on Ubiquitous Computing*, pages 246–263, London, UK, 2002. Springer-Verlag.

[14] Tim Kindberg, John Barton, Jeff Morgan, Gene Becker, Debbie Caswell, Gita Gopal, Marcos Frid, Venky Krishnan, Howard Morris, John Schettino, and Bill Serra. People, places, things: web presence for the real world. In *In proceedings WMCSA2000. Available as http://www.cooltown.hp.com/papers/webpres/webpresence.htm*, pages 365–376, 2000.

[15] J. Kunze. Encoding dublin core metadata in html, 1999.

[16] Ray R. Larson and Patricia Frontiera. Ranking and representation for geographic information retrieval.

[17] Per Persson, Fredrik Espinoza, Petra Fagerberg, Anna Sandin, and Rickard Cöster. Geonotes: a location-based information system for public spaces. pages 151–173, 2003.

Setup

This chapter covers the software and library dependencies of the components of this system and depicts the database layouts used by the web applications.

A.1 Databases

PostgreSQL was used for storing data of the distinct web applications. The database layouts of the *frontend*, the *webcrawler/search engine* and the *VBZ service* can be seen in figures A.1, A.2 and A.3 respectively.

To be able to use the PostGIS extension for PostgreSQL in a reasonable way, a database template can be generated (see listing A.1). The databases using the PostGIS extension (*frontend* and *webcrawler*) can then be created using the "template_postigs" template.

```
$ psql template1
\c template1
CREATE DATABASE template_postgis with template = template1;

-- set the 'datistemplate' record in the 'pg_database' table for
-- 'template_postgis' to TRUE to indicate that it is a template
UPDATE pg_database SET datistemplate = TRUE where datname = 'template_postgis';
\c template_postgis
CREATE LANGUAGE plpgsql;
\i /usr/share/postgresql/contrib/lwpostgis.sql
\i /usr/share/postgresql/contrib/spatial_ref_sys.sq

-- set role based permissions in production env.
GRANT ALL ON geometry_columns TO PUBLIC;
GRANTALL ON spatial_ref_sys TO PUBLIC;

-- vacuum freeze: it will guarantee that all rows in the database are
-- "frozen" and will not be subject to transaction ID wraparound
-- problems.
VACUUM FREEZE;
```

Listing A.1: Creation of the "template_postgis" database template

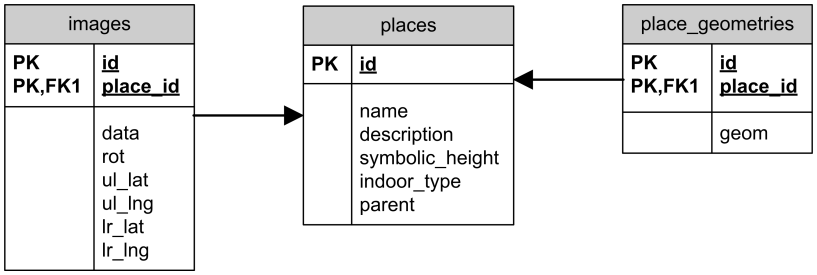


Figure A.1: Database layout used by the *frontend*

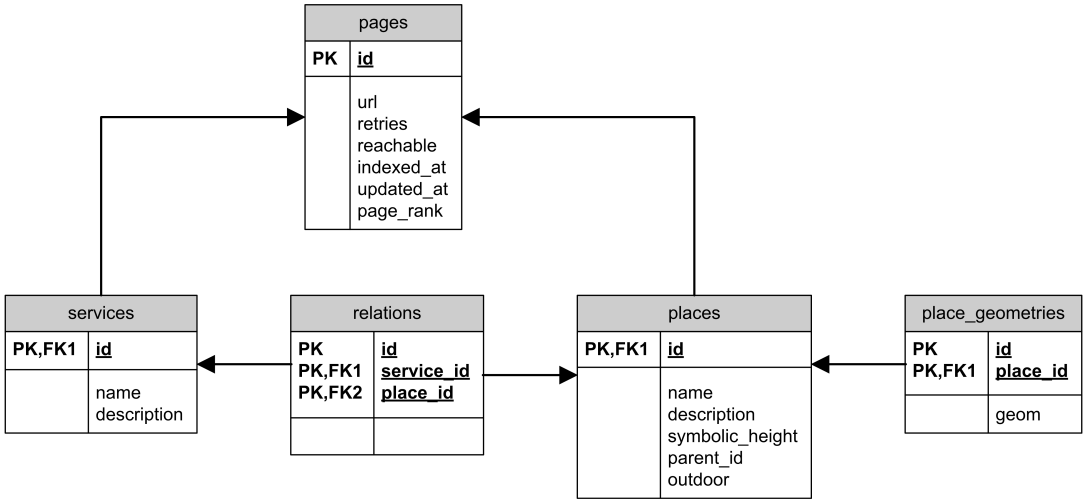


Figure A.2: Database layout used by the *webcrawler* and the *search engine*

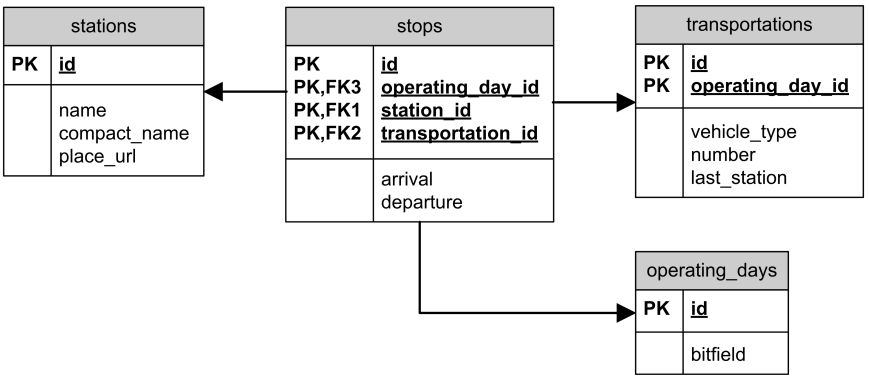


Figure A.3: Database layout used by the *VBZ service*

A.2 Webcrawler

An Apache ant file has been provided with the *webcrawler* that can be used to build an executable jar file containing all required libraries (package-all target). Listing A.2 shows the provided targets.

```
$ ant -p

Buildfile: build.xml

Main targets:

 clean       Cleans the target-directory
 compile     Compiles the project
 copy-metafiles Copies all metafiles into the builddirectory
 copy-resources Copies all resources from sourcedirectory to the builddirectory
 package     Creates a jar-package of the project
 package-all Creates a jar-package containing all required libraries
 run         Compiles and executes the web-crawler
Default target: compile
```

Listing A.2: Ant targets of the *webcrawler*

This ant file, for example, can be used for adjusting the *webcrawler* properties via the user.properties file located in the resources directory of the *webcrawler* project path.
The executable jar file should be placed in the "external" directory of the *search engine*'s RoR project root, since the *search engine* will invoke the *webcrawler* to index newly detected or modified web resources.

The required libraries are listed below:

Library	**Description**
log4j-1.2.11.jar	Logging facility.
postgresql-8.3-603.jdbc3.jar	PostgreSQL database connector.
postgis_1.3.2.jar	Used for being able to store geometric datatypes in the PostgreSQL database.

A.3 Mobile Phone Client

As the *mobile phone client* was written in Python, it requires a Python environment to be installed together with a Python script shell. This client is provided as a single Python script that can be copied onto the mobile phone and executed using the Python script shell. However, this script shell needs to be signed, since the mobile phone's GPS capabilities can not be used by unsigned components; the same applies to the used Bluetooth library.

The follwing software components need to be installed on the mobile phone:

Library	**Description**
PythonForS60_1_4_4_3rdEd.sis	Python interpreter.
PythonScriptShell_1_4_3_3rdEd.sis (signed)	Python script shell used to execute Python scripts.
lightblue-0.3.2-s60-3rdEd.sis (signed)	Used for Bluetooth device sensing.

A.4 RoR Projects

Each of the RoR projects comes with an integrated web server. These servers can be started when being in the command line inside the projects root directory by invoking:

```
./script/server
```

However, it is recommended to use a server like Apache to run the individual web applications.

The databases used by the individual RoR web applications must be setup and running beforehand, as the RoR application will terminate otherwise. The database connections are configured in the database.yml file located in the config directory of the RoR projects.

Used Versions

Following a list of software components used for developing the RoR web applications. Each of the components is categorized depending on how it has been installed which can be one of the following:

gem A gem is a software package, containing a Ruby library, for instance. Gems can be installed by using RubyGems, a package manager used for administrating available gems. An installed gem is usable by all RoR projects.

plug-in Plug-ins are installed for individual RoR projects by invoking:

```
./script/plugin install <SVN_path>
```

Since plug-ins are taken from SVN the revision number is listed besides the plug-in in the table below. The SVN locations of the used plug-ins are:

```
ym4r_gm: SVN://rubyforge.org/var/SVN/ym4r/Plugins/GM/trunk/ym4r_gm
spatial_adapter: svn://rubyforge.org/var/svn/georuby/SpatialAdapter/trunk/spatial_adapter
```

system This means that the needed software components should be installed using the underlying system (e.g., Linux).

Component	Inst.Type	Used by	Description
rails-2.0.2 rake-0.8.1 activesupport-2.0.2 activerecord-2.0.2 actionpack-2.0.2 actionmailer-2.0.2 activeresource-2.0.2	**gem**	all	The Ruby On Rails base components.
ruby-postgres-0.7.1.2006.04.06	**gem**	all	RoR PostgreSQL bindings.
GeoRuby-1.3.3	**gem**	frontend, search engine	GeoRuby provides geometric data types from the OGC "Simple Features" specification.
haml-1.8.2	**gem**	all	Haml can be used to parse view templates written in the Haml markup language.
piston-1.4.0	**gem**	all	RoR plug-in management utility.
rmagick-2.5.2	**gem**	frontend	Ruby ImageMagick bindings. Used for manipulation of images.
spatial_adapter-r116	**plug-in**	frontend, search engine	A plug-in for Rails which manages PostGIS geometric columns in a transparent way. Depends on GeoRuby.
ym4r_gm-r103	**plug-in**	frontend	Used to facilitate the use of Google Maps from a RoR application.
imagemagick-R6.4.0.6	**system**	frontend	ImageMagick is used for the manipulation of images.
ruby-R1.8.6_p287-r1	**system**	all	Ruby language.

Configuration

B

B.1 Webcrawler

The *webcrawler* provides a set of properties used to configure database related settings and properties that can be used to adjust its behavior with respect to crawling web pages. The default properties are defined in the system.properties file in the resources directory. These properties can be overridden by a user.properties file that should be placed in the same directory as the system.properties file.The default properties are listed in listing B.1 and explained below.

```
crawler.initialLink = localhost
crawler.name = wopaloC
crawler.request.timeout = 60000
crawler.request.sleepTime = 20000
crawler.request.recrawlDelay = 14400000
crawler.db.url = jdbc:postgresql://localhost:5432/wopalo_index
crawler.db.username = wopalo
crawler.db.password = wopalo
```

Listing B.1: The system.properties file of the *webcrawler*

crawler.initialLink Defines the URL of the resource which will be crawled if no command line parameters have been provided and no *places* or *services* are indexed yet.

crawler.name Name of the *webcrawler* which will be used as user-agent parameter when doing requests to web resources.

crawler.request.timeout Timeout to use when trying to connect to a web server or download a web resource.

crawler.request.sleepTime Time to sleep before processing the next web resource.

crawler.request.recrawlDelay Timeout to use before re-crawling already indexed web resources. This can be used to get behavior like: re-crawl the resource at most every 2 days.

crawler.db.* Database specific settings.

B.2 Search Engine

The *search engine* provides a set of properties that can be set in the environment.rb file in the configuration directory of *search engine*'s RoR project root. The important ones are the following:

MASK_VERTICAL Height of the *search mask* in terms of longitude.

MASK_HORIZONTAL Height of the *search mask* in terms of latitude.

PLACES_VIEW_COUNT Determines the maximum number of *places* to return for a *spatial query*.

SERVICES_VIEW_COUNT Determines the maximum number of *services* to return when querying for *services* accessible at a given *place*.

B.3 Mobile Phone Client

The properties of the *mobile phone client* are defined at the beginning of the respective Python script. A example is shown in listing B.2; the most important properties are explained below:

```
HTTP_PREFIX = u'http://'
HOST = HTTP_PREFIX + u'129.132.75.250:4000'
PLACES_SEARCH_PATH = HOST + u'/places.compact?'
SERVICES_SEARCH_PATH = HOST + u'/services.compact?'
MAX_GPS_QUERY_TIME = 35
MAX_BLUETOOTH_SENSING_TIME = 35
MAPPINGFILE = u'e:\\wopalo.map'
TEMPFILE = u'e:\\wopalo.tmp'
LOGFILE = u'e:\\wopalo.log'
PLACES_VIEW_COUNT = 8
SERVICES_VIEW_COUNT = 8
```

Listing B.2: Constants of the *mobile phone client*

HOST The host where the *search engine* is accessible at.

MAX_GPS_QUERY_TIME Defines the timeout after which an ongoing GPS inquiry is aborted.

MAX_BLUETOOTH_SENSING_TIME Defines the timeout after which ongoing Bluetooth sensing is aborted.

MAPPINGFILE Determines the location of the file that associates Bluetooth hardware addresses with *indoor positions*.

PLACES_VIEW_COUNT The number of *places* to display at once. Used for pagination.

SERVICES_VIEW_COUNT The number of *services* to display at once. Used for pagination.

B.4 Frontend

The *frontend* requires a configured HOST parameter, specifying the host the RoR application is running on. This parameter has to be set in the environment.rb file located in the configuration directory and in the application.js file located in the public/javascript directory.
Additionally, a GoogleMaps key must be generated for the the host the application is running on and be assigned to the GMAPS_HOST parameter in the environment.rb file and set in the gmaps_api_key.yml file located in the configuration directory. The key is required as otherwise no map will be displayed when accessing the *frontend*.

Ruby On Rails Server API

This section covers the possible parameters that can be used when invoking different RoR routes that are part of the public API. Parameters for the following routes are annotated with a "req", whereas optional parameters are annotated with "opt". The routes are covered separately for the individual web applications. For every route a valid sample invocation is shown at the beginning.

C.1 Search Engine

`GET /places(.:format)`

```
GET /places.compact?p=47.380284717,8.5447883605&count=8&from=0&filter=contained
```

This route is used for spatial queries and accepts the following parameters:

p (req) A spatial position encoded as latitude/longitude pair using WGS84 representation

h (opt) Integer value specifying the floor the user is located on. Providing this parameter will tell the *search engine* to filter *places* which are located outdoors.

ne (opt) North-east spatial position of a custom set search box; encoded as latitude/longitude in WGS84. This parameters requires the "sw" parameter to be set.

sw (opt) South-west spatial position of a custom set search box; encoded as latitude/longitude in WGS84. This parameters requires the "ne" parameter to be set.

filter (opt) If set to "contained" only *places* the provided spatial position is contained in will be returned.

from (opt) Integer value specifying the offset of a custom result mask. This can be used for pagination.

count (optl) Number of results that should be returned. This can be used for pagination.

Besides these parameters this route is capable of returning results in two different formats: HTML or compact. The requested format can be specified when invoking the route. The first of the two server requests listed below will return the results in HTML format, the third will return the results in compact format which is of MIME type text/plain.

```
GET /places?p=47.3802847176866,8.54478836059571
GET /places.html?p=47.3802847176866,8.54478836059571
GET /places.compact?p=47.3802847176866,8.54478836059571
```

The result returned in compact format has the following form:

```
number_of_places_found,result_mask_offset,result_mask_count
place_name,place_description,place_url
place_name,place_description,place_url
                :
```

A typical result looks like follows:

```
25,1,8
ETHZ IFW,ETH Informatikgebäude,http://129.132.75.250:3000/places/17
IFW Dachterasse,Dachterasse IFW Gebäude ETH Zürich,http://129.132.75.250:3000/places/132
RZ-Building,RZ-Building of ETH Zurich,http://129.132.75.250:3000/places/32
Gangway between IFW and RZ,Connection between IFW and RZ buildings,http://129.132.75.250:3000/places/33
Test Building,Test of an indoor building,http://129.132.75.250:3000/places/24
Crosswalk,Crosswalk,http://129.132.75.250:3000/places/135
Church of Liebfrauen,http://de.wikipedia.org/wiki/Liebfrauenkirche_...,http://129.132.75.250:3000/places/28
Haldenegg,Trambahnhaltestelle Haldenegg,http://129.132.75.250:3000/places/22
```

`GET /services(.:format)`

```
GET /services.compact?url=http://places.wopalo.org/places/17&from=2
```

This route can be used to search for *services* that reference the *place* with the provided URL. The route accepts the following parameters:

url (req) The URL of the *place* for which to retrieve *services*.

from (opt) Integer value specifying the offset of a custom result mask. This can be used for pagination.

count (opt) Number of results that should be returned. This can be used for pagination.

The route can return results in HTML or compact format. The result returned when requesting services in compact format has the following form:

```
number_of_services_found,result_mask_offset,result_mask_count
service_name,service_description,service_url
service_name,service_description,service_url
                :
```

A typical result would look as follows:

```
1,1,1
VBZ Fahrplaninfo,Anschlüsse ab Zürich Haldenegg,http://vbzservice.wopalo.org/station/299915
```

C.2 Frontend

`GET /map.viewport(.:format)`

```
GET /map/viewport.kml?ne=47.39;8.58&sw=47.36;8.50&blacklist=1;81;83;23;41
```

This route can be used to retrieve *places* hosted by the *frontend* and contained in a provided viewport. It accepts the following parameters:

ne (req) North-east spatial position of the viewport; encoded as latitude/longitude in WGS84.

sw (req) South-west spatial position of the viewport; encoded as latitude/longitude in WGS84.

blacklist (opt) A semicolon separated list of *place* ids for *places* which should not be returned in the result.

This route can return results in the following formats: GML or JSON. If no format is specified the result will be returned in JSON.

GET /places/:id/image

```
GET /places/18/image?rot=338.5946917230823&scale=0.27128718187291617
```

This route requires an id to be provided which identifies the *place*. If no other parameters besides the id are provided the route returns the image belonging to the *place* in its original size. However, the route accepts optional parameters:

rot Float value determining the angle by which the image should be rotated. The default value is $0°$. The image will be rotated clockwise.

scale Float value determining the factor by which the image should be scaled. The default value is 1.0.

width Integer value specifying the width of the returned image in pixels.

height Integer value specifying the height of the returned image in pixels.

If the image should be both rotated and scaled, it is first rotated and then scaled.